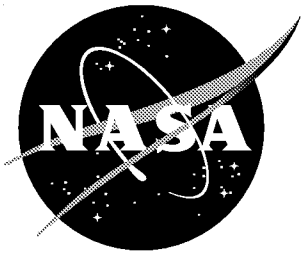
NASA/CR-1999-208992



Formal Verification of the AAMP-FV Microcode

Steven P. Miller, David A. Greve, Matthew M. Wilding
Rockwell Collins, Cedar Rapids, IA

Mandayam Srivas
SRI International, Menlo Park, CA

February 1999

The NASA STI Program Office ... in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA Scientific and Technical Information (STI) Program Office plays a key part in helping NASA maintain this important role.

The NASA STI Program Office is operated by Langley Research Center, the lead center for NASA's scientific and technical information. The NASA STI Program Office provides access to the NASA STI Database, the largest collection of aeronautical and space science STI in the world. The Program Office is also NASA's institutional mechanism for disseminating the results of its research and development activities. These results are published by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services that complement the STI Program Office's diverse offerings include creating custom thesauri, building customized databases, organizing and publishing research results ... even providing videos.

For more information about the NASA STI Program Office, see the following:

- Access the NASA STI Program Home Page at ***http://www.sti.nasa.gov***

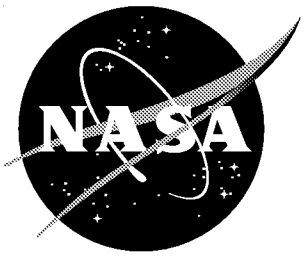
- E-mail your question via the Internet to help@sti.nasa.gov

- Fax your question to the NASA STI Help Desk at (301) 621-0134

- Phone the NASA STI Help Desk at (301) 621-0390

- Write to:
  NASA STI Help Desk
  NASA Center for AeroSpace Information
  7121 Standard Drive
  Hanover, MD 21076-1320

NASA/CR-1999-208992



Formal Verification of the AAMP-FV Microcode

Steven P. Miller, David A. Greve, Matthew M. Wilding
Rockwell Collins, Cedar Rapids, IA

Mandayam Srivas
SRI International, Menlo Park, CA

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

Prepared for Langley Research Center
under Contracts NAS1-19704 and NAS1-20334

February 1999

Available from:

NASA Center for AeroSpace Information (CASI)
7121 Standard Drive
Hanover, MD 21076-1320
(301) 621-0390

National Technical Information Service (NTIS)
5285 Port Royal Road
Springfield, VA 22161-2171
(703) 605-6000

Contents

1 Introduction **1**

2 Background **3**
2.1 NASA Langley, SRI International, and Collins 3
2.2 The AAMP Family of Microprocessors 3
2.3 PVS 4
2.4 Related Work 5
2.5 Formal Verification of the AAMP5 5
2.6 Overview of Processor Correctness 6

3 Project Goals and History **8**
3.1 Project Goals 8
3.2 Project History 9

4 The Macroarchitecture: The Programmer's View of the AAMP-FV **13**
4.1 Overview of the AAMP-FV Macroarchitecture 13
4.1.1 Organization of Memory 13
4.1.2 Process Stack 13
4.1.3 Stack Cache 15
4.1.4 Internal Registers 15
4.1.5 Instruction Set and Data Types 16
4.1.6 Multi-Tasking and Error Handling 16
4.2 Formal Specification of the Macroarchitecture 18
4.2.1 Bit Vectors 19
4.2.2 Memory 20
4.2.3 Macroarchitecture State 20
4.2.4 Next Macro State Function 22

5 The Microarchitecture: The Register Transfer View of the AAMP-FV **25**
5.1 Overview of the AAMP-FV Microarchitecture 25
5.1.1 The Data Path 25
5.1.2 The Microcontroller 27
5.1.3 The Bus Interface Unit 28
5.2 Formal Specification of the Microarchitecture 28
5.3 Formal Specification of the Microcode 31

6 Formal Verification of the AAMP-FV **33**
6.1 Overview . . . 33
6.1.1 Commutativity Theorems . . . 33
6.1.2 Visibility Theorems . . . 34
6.1.3 Invariant Theorems . . . 34
6.2 The Micro Correctness Proofs . . . 34
6.2.1 Standard AAMP-FV Instructions . . . 35
6.2.1.1 The Micro Correctness Theory . . . 35
6.2.1.2 The Micro Correctness Proofs . . . 38
6.2.1.3 Proofs of Visibility Properties . . . 39
6.2.1.4 Proofs of Invariant Properties . . . 39
6.2.2 The Complex AAMP-FV Instructions . . . 39
6.2.2.1 The CALL Instruction . . . 40
6.3 Proof of the Stack Adjustment Logic . . . 43
6.4 The Macro Lift Proofs . . . 43
6.4.1 The Abstraction Function . . . 43
6.4.2 The Macro Correctness Statement . . . 43
6.4.3 The Macro Lift Proofs . . . 45

7 Lessons Learned **47**
7.1 Technology Transfer . . . 47
7.2 Development of Domain Specific Libraries . . . 48
7.3 Proof Robustness . . . 49
7.4 Exploiting Modularity . . . 50
7.5 Support for Product Families . . . 51
7.6 Importance of the User Interface . . . 52
7.7 What Needs to be Proven? . . . 53
7.8 Support for Team Efforts . . . 54
7.9 Use of Human Resources . . . 55

8 Conclusions and Future Directions **57**

List of Figures

2.1 Pictoral Representation of Microcode Correctness 6

4.1 The Process Stack 14
4.2 Macroarchitecture Specification Hierarchy 18
4.3 PVS Specification of AAMP-FV Bit Vectors 19
4.4 PVS Specification of AAMP-FV Memory 21
4.5 PVS Specification of AAMP-FV Macroarchitecture State 22
4.6 PVS Specification of REFA instruction 23

5.1 The AAMP-FV Microarchitecture 26
5.2 PVS Specification of CONNECT 30
5.3 PVS Specification of the Next PC Register 30
5.4 PVS Specification of the REFA Microcode 32

6.1 Overview of the Correctness Proof 33
6.2 CALL Proof Structure 41
6.3 Step Function 42
6.4 Abstraction Function 44
6.5 PVS Correctness Statement for the REFA Instruction 45

List of Tables

2.1 Applications of the CAPS/AAMP Family 4

3.1 Level of Effort 10
3.2 Proofs Completed 11

Chapter 1

Introduction

Software and digital hardware are increasingly being used in safety-critical systems such as aircraft, nuclear power plants, weapon systems, and medical instrumentation. Several authors have demonstrated the infeasibility of showing that such systems meet ultra-high reliability requirements through testing alone [8, 26]. Formal specification combined with mechanical proofs of correctness are a promising approach for achieving the extremely high levels of assurance required of safety-critical systems, but there have been few examples of the use of such approaches in industry.

Previous papers have described the formal verification of the microcode in a Rockwell proprietary microprocessor, the AAMP5 [27, 37, 35, 38, 36]. Sponsored by the Assessment Technology Branch of NASA Langley and Collins Commercial Avionics, a division of Rockwell International, this project was conducted by Collins and SRI's Computer Science Laboratory. The project consisted of specifying in the PVS language developed by SRI [28, 29] a portion of a Rockwell proprietary microprocessor, the AAMP5, at both the instruction set and register-transfer levels and using the PVS theorem prover to show the microcode correctly implements the specified behavior for a representative subset of instructions.

The central result of the AAMP5 project was to demonstrate that formal verification of the microcode for a large, pipelined microprocessor was technically feasible. Over half the AAMP5 instruction set was formally specified at the macroarchitecture level, and all of the microarchitecture needed for formal verification of the microcode was specified. The microcode for eleven instructions, representative of several instruction classes, was proven correct in the absence of interrupts. Another key result was the discovery of both actual and seeded errors.

However, the AAMP5 project was very much an exploratory project, and the cost to verify each instruction was quite high. While it was clear that costs could be reduced significantly on the next project, there was no way to accurately estimate how large this savings would be. Thus the AAMP5 project left unanswered the question of whether formal verification of microcode could be performed in a cost effective manner. To address this question, NASA, SRI, and Collins decided to repeat the experiment with a new processor, the AAMP-FV, to see if the expertise gained during the AAMP5 project could be used to bring the cost of down to an acceptable level.

Rather than choose a microprocessor currently under development, as was done with the AAMP5, the decision was made to apply formal verification to a processor specifically designed for use in ultra-critical applications. The AAMP-FV is a paper and pencil design of a processor specifically designed for use in applications such as autoland or fly-by-wire. As a result, it is simpler than other members of the AAMP family, though it is by no means a toy. Like all members of the

AAMP family, it is a stack-based machine designed for use with block-structured, high-level languages such as Ada in real-time embedded applications, and provides hardware support for many features normally provided by the compiler and the run-time environment.

The key result of the AAMP-FV project is to confirm that the expertise gained on the AAMP5 project can be exploited to reduce the cost of formal verification dramatically. Of the 80 AAMP-FV instructions, 54 were proven correct. More importantly, the cost of their verification dropped by almost an order of magnitude from that observed on the AAMP5 project. In many ways, verification of these 54 instructions typified a true engineering process, using well understood methods to achieve clearly defined goals in the expected amount of time.

However, this was not the case for the entire project. As more complex instructions were attempted, proof techniques first developed on the AAMP5 project broke down and new approaches had to be devised. This phase progressed more as an exploratory project, with a steep learning curve and unexpected delays. While fewer instructions were verified during this phase, several important new techniques were developed. One of the main contributions of the AAMP-FV project was the development of methods to handle instructions with complex microcode.

Organization of the Report

This report is organized as follows. Chapter 2 provides general background, describing the participants in the project, the history of the AAMP family of microprocessors, the PVS specification language, and a brief survey of related work. Chapter 3 discusses the goals and history of the project. Chapter 4 describes the AAMP-FV instruction set (macro) architecture and its specification in PVS. Chapter 5 provides a similar discussion of the AAMP-FV register transfer (micro) architecture. Chapter 6 describes the formal verification effort. Chapter 7 discusses lessons learned on both the AAMP5 and AAMP-FV projects, and chapter 8 summarizes our conclusions and suggestions for future work.

Chapter 2

Background

The following sections discuss the AAMP family of microprocessors, the PVS verification system, related work, and provide a brief overview of the technical approach.

2.1 NASA Langley, SRI International, and Collins

NASA Langley's research program in formal methods [9] was established to bring formal methods technology to a sufficiently mature level for use by the United States aerospace industry. Besides the inhouse development of a formally verified reliable computing platform RCP [14], NASA has sponsored a variety of demonstration projects to apply formal methods to critical subsystems of real aerospace computer systems.

The Computer Science Laboratory of SRI International has been involved in the development and application of formal methods for more than twenty years. The formal verification systems EHDM and PVS were both developed at SRI. Both EHDM and PVS have been used to perform several verifications of significant difficulty, most notably in the field of fault-tolerant architectures and hardware designs. Recently, SRI has been actively involved in investigating ways to transfer formal verification technology to industry.

Collins Avionics & Communications is a division of Rockwell International and one of the largest suppliers of communications and avionics systems for commercial transport and general aviation aircraft. Collins' interest in formal methods dates from 1991 when it participated in the MCC Formal Methods Transition Study [17]. As a result of this study, Collins initiated several small pilot projects to explore the use of formal methods, including formal verification of the AAMP5 [27, 37, 35, 38, 36].

2.2 The AAMP Family of Microprocessors

The Advanced Architecture Microprocessor (AAMP) is a Rockwell proprietary family of microprocessors based on the Collins Adaptive Processing System (CAPS) originally developed in 1972 [3]. The AAMP architecture is specifically designed for use with block-structured, high-level languages such as Ada in real-time embedded applications. It is based on a stack architecture and provides hardware support for many features normally provided by the compiler and run-time environment, such as procedure state saving, parameter passage, return linkage, and reentrancy. The AAMP

also simplifies real-time executive design by implementing in hardware such functions as interrupt handling, task state saving, and context switching. Use of internal registers holding the top few elements of the stack provides the AAMP family with performance that rivals or exceeds that of most commercially available 16-bit microprocessors.

The original CAPS architecture, a multiboard minicomputer, was developed in 1972 and was quickly followed by the CAPS-2 through CAPS-10. In 1981, the original AAMP consolidated all CAPS functions except memory on a single integrated circuit. It was followed by the AAMP2, AAMP3, and AAMP5. Members of the CAPS/AAMP family have been used in an impressive variety of products as shown in Table 2.1.

Table 2.1: Applications of the CAPS/AAMP Family

CAPS-4	1974	Global Positioning System, General Development Model (GPS GDM)
CAPS-6	1977	Boeing 757, 767 Autopilot Flight Director System (AFDS) Lockheed L-1011 Active Control System (ACS) Lockheed L-1011 Digital Flight Control System (DFCS) NASA Fault Tolerant Multiprocessor (FTMP)
CAPS-8	1979	Boeing 757, 767 Electronic Flight Instrumentation System (EFIS) Boeing 757, 767 Engine Instrumentation/Crew Alerting System (EICAS)
CAPS-7	1979	Navstar Global Positioning System (GPS) Boeing 747-400 Integrated Display System (IDS)
CAPS-10	1979	Boeing 747-400 Central Maintenance Computer (CMC) Boeing 737-300 Electronic Flight Instrumentation System (EFIS)
AAMP1	1981	Boeing 737-300 Engine Instrumentation/Crew Alerting System (EICAS) Air Transport Traffic Collision Avoidance System (TCAS)
AAMP2	1987	Air Transport TCAS Vertical Speed Indicator (TVI) Boeing 777 Flight Control Backdrive Commercial GPS: Navcore I, Navcore II, Navcore V
AAMP3	1992	Boeing 777 Standby Instruments
AAMP5	1993	Global Positioning Systems, Upgrade for AAMP2

2.3 PVS

PVS (Prototype Verification System) [31] is an environment for specification and verification that has been developed at SRI International's Computer Science Laboratory. In comparison to other widely used verification systems, such as HOL [19] and the Boyer-Moore prover [5], the distinguishing characteristic of PVS is that it supports a highly expressive specification language with a very effective interactive theorem prover in which most of the low-level proof steps are automated. The system consists of a specification language, a parser, a type checker, and an interactive proof checker. The PVS specification language is based on higher-order logic with a richly expressive type system so that a number of semantic errors in specification can be caught during type checking. The PVS prover consists of a powerful collection of inference steps that can be used to reduce a proof goal to simpler subgoals that can be discharged automatically by the primitive proof steps

of the prover. The primitive proof steps involve, among other things, the use of arithmetic and equality decision procedures, automatic rewriting, and BDD-based Boolean simplification.

2.4 Related Work

Microprocessor and microcode verification is not new. A number of microprocessor designs have been formally verified [2, 11, 12, 20, 41]. Microcode verification was pioneered by Bill Carter [25] at IBM in the 1970's and applied to elements of NASA's Standard Spaceborne Computer [25]; in the 1980's a group at the Aerospace Corporation verified microcode for an implementation of the C/30 switching computer using a verification system called SDVS [12]; and a group at Inmos in the UK established correctness across two levels of description (in Occam) of the microcode for the T800 floating-point unit using mechanized transformations [1].

Several groups have performed automated verification of non-microcoded processors, of which Warren Hunt's FM8501 [20] and subsequent FM9001 [21] are among the most substantial. The problems of pipeline correctness were studied previously by Srivas and Bickford [34], by Saxe and Garland [30], Burch and Dill [7], and Windley and Coe [42]. A very simple microcoded processor design developed by Mike Gordon called "Tamarack" serves as something of a benchmark for microprogram verification and was considered quite a challenge not so long ago [22]. PVS is able to verify the microcode of Tamarack completely automatically in about five minutes [13].

Other projects have used automatic theorem provers to reason about programs written in low-level languages. A simple machine is described precisely and the machine code that implements an operating system kernel is proved to implement correctly several properties needed of an operating system kernel [4]. Most of the instructions of a 68020 processor have been formalized and some C subroutines compiled to the 68020 are specified and checked mechanically [6]. Some programs written for the FM9001 have been proved correct using a theorem prover, including some proved to achieve desired real-time system properties [39, 40]. Another formalized processor, the MIPS R3000 [23], has had the implementation of a round-robin scheduler proved correct using a theorem prover [16]. Such proofs of low-level programs have distinct goals from the AAMP-FV effort, as none verified the correct operation of microcoded instructions. However, there are many similarities in the structure of the underlying proofs.

2.5 Formal Verification of the AAMP5

The AAMP-FV project grew out of an earlier effort to verify formally the microcode in another Rockwell microprocessor, the AAMP5 [27, 37, 35, 38, 36]. The AAMP5 was designed to be object code compatible with the earlier AAMP2 while providing a threefold improvement in throughput. The AAMP2 was developed as a general purpose microprocessor for use in a variety of avionics displays and global positioning systems. It has a large, complex instruction set, supports a variety of data types, implements in hardware many of features needed to support high-level block structured languages, and provides extensive support for multi-tasking and error handling.

To obtain a threefold improvement in performance, the AAMP5 implemented the AAMP2 instruction set using internal pipelining and look ahead fetching of both instructions and data. Since it provides high performance in a general purpose processor, the AAMP5 is one of the most complex microprocessors to which formal methods have been applied. One measure of the

complexity of a processor is the size of its implementation. The AAMP5 contains some 500,000 transistors, as compared to some tens of thousands in previous formally verified designs and 3.1 million in an Intel Pentium.

Even so, the AAMP5 project succeeded in demonstrating the technical feasibility of the approach. Over half the AAMP5 instruction set was formally specified at the macroarchitecture level, and all of the microarchitecture needed for formal verification of the microcode was formally specified. The microcode for eleven instructions, representative of several instruction classes, was proven correct in the absence of interrupts.

Another key result was the discovery of both actual and seeded errors. Two actual microcode errors were discovered during development of the formal specification and removed before first fabrication of the microprocessor, illustrating the value of simply creating a precise specification. Two additional errors seeded by Collins in the microcode were systematically uncovered by SRI while doing correctness proofs. One of these was an actual error that had been discovered by Collins after first fabrication but left in the microcode provided to SRI. The other error was designed to be unlikely to be detected by walk-throughs, testing, or simulation.

Several other results emerged during the project, including the ease with which practicing engineers became comfortable with PVS, the need for libraries of general-purpose theories, the usefulness of formal specification in revealing errors, the natural fit between formal specification and inspections, the difficulty of selecting the best style of specification for a new problem domain, the high level of assurance provided by proofs of correctness, and the need to engineer proof strategies for reuse.

2.6 Overview of Processor Correctness

The verification of a microprocessor normally involves specifying the processor as a machine that executes instructions at two levels — the macro and the micro level — and then proving a desired correctness condition that relates the behavior of the processor at these two levels. The macro level specification describes the effect of executing an instruction on the state visible to an assembly language programmer. The micro level specification describes the processor at the register-transfer level, defining the effect of executing an arbitrary microinstruction on the movement of data between the registers and other components in the processor's design.

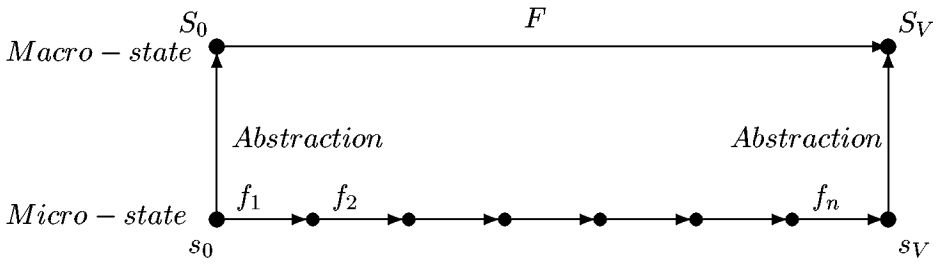


Figure 2.1: Pictoral Representation of Microcode Correctness

Figure 2.1 represents the correct operation of a microcoded instruction. Three kinds of instruction execution properties are used to formalize the desired processor behavior.

- The most important correctness properties are the *commutativity* theorems illustrated in Figure 2.1. These consist of showing that the sequence of microinstructions $f_1, f_2, ..., f_n$ making up each machine instruction F causes a corresponding change in the micro-state s_0 as F does to the macro-state S_0. This is done by defining a function *Abstraction* that maps elements of the micro state to elements of the macro-state and proving that $F(Abstraction(s_0)) = Abstraction(f_n(...(f_2(f_1(s_0)))...))$.

- Several assumptions about the initial micro state s_0 are needed to show that the commutativity theorems hold. A micro-state that satisfies these assumptions is called a *visible* state, represented in Figure 2.1 with larger circles. Showing that the final micro-state of a microcoded instruction's execution (micro-state s_V) is also a visible state ensures that execution of the microcode for each machine instruction leaves the processor in the proper state for the next machine instruction. We call correctness statements of this kind *visibility* theorems.

- Other necessary correctness properties include invariants on each of the sequence of micro-states $s_0...s_V$ during execution of an instruction. We call correctness theorems of this kind *invariant* theorems.

Chapters 4 and 5 describe in greater detail the AAMP-FV macro and microarchitectures and their specification in PVS. Chapter 6 discusses the theorems proved about AAMP-FV microcode.

Chapter 3

Project Goals and History

This chapter discusses the motivation, tasks, and level of effort devoted to the AAMP-FV project.

3.1 Project Goals

The main goal of the AAMP-FV project was to determine if formal verification of microcode could be performed in a cost effective fashion for a processor designed for use in ultra-critical applications. While the earlier AAMP5 project succeeded in demonstrating the technical feasibility and value of formal verification, the cost was quite high. Simply dividing the total project hours by the number of instructions verified resulted in a figure of 308 hours per instruction.

Those close to the project understood that this greatly exaggerated the true cost. The AAMP5 project was highly exploratory, making it difficult to determine what portion of the project cost should be attributed to mastering a new technology and what portion would be incurred using the same approach on future projects. Large parts of the project were devoted to the development of supporting libraries, such as the bit vectors, and considerable time was spent by SRI in becoming familiar with the AAMP5 and by Collins in mastering PVS. In addition, only a few instructions in each class were verified before moving on to the next class; far more instructions could have been completed if only one or two classes had been attempted. Finally, time simply ran out before a large number of the proofs could be completed, even though much of the necessary infrastructure had been put in place.

To get a better estimate of the true cost of formal verification of microcode, NASA, SRI, and Collins decided to repeat the experiment with a different processor, the AAMP-FV, to determine if the expertise gained during the AAMP5 project could be used to bring the cost of formal verification down to an acceptable level.

An advantage of the AAMP5 project was that formal methods were applied in parallel with the development of an actual microprocessor. However, this also had its drawbacks. The size and complexity of the AAMP5 made it a formidable example for mastering formal methods. Also, while intended for critical applications such as avionics displays and global positioning system, the AAMP5 was not developed for use in the most critical applications such as autoland and fly-by-wire. At the conclusion of the AAMP5 project, it was generally felt that if a follow-on effort were to be undertaken, it should focus on a processor specifically designed for use in ultra-critical applications.

Unfortunately, an actual processor meeting these criteria was not scheduled for development in the near future.

As a result, the the AAMP-FV is a paper and pencil design of a processor representative of one that would be used in ultra-critical applications such as autoland or fly-by-wire. To make it easier to verify, either by traditional methods or by formal methods, it is simpler than other members of the AAMP family. It has a smaller instruction set, fewer data types and addressing modes, a flat address space, is not pipelined, and prefetches only in that reads are performed a word at a time and a word may contain up to two instructions. Even so, the AAMP-FV is not a toy design. If fabricated, it would contain approximately 100,000 transistors, as compared to some 500,000 transistors in the AAMP5.

3.2 Project History

The main activities of the AAMP-FV project and the level of effort invested in each are shown in Table 3.1. Unlike the AAMP5 project, the specification in PVS of the macroarchitecture (instruction set) and microarchitecture (register transfer level) was well understood by the time the AAMP-FV project was started and was done almost entirely by Collins. Specification of the AAMP-FV macroarchitecture has taken 130 hours to date and consists of 3,764 lines of PVS covering 54 instructions. In contrast, development of the AAMP5 macroarchitecture specification took 941 hours, consisted of 2,550 lines of PVS, and covered 108 instructions [27]. Specification of the AAMP-FV microarchitecture took only 90 hours and consists of 3,496 lines of PVS, as compared to approximately 1,100 hours and 2,679 lines of PVS for the AAMP5 [27].

This significant (almost an order of magnitude) reduction in the time to specify the micro and macro architectures occurred because of the experience gained on the AAMP5, reuse of existing libraries, and the simpler architecture of the AAMP-FV. Of these, we believe the experience gained and the reuse of existing libraries, particularly the bit vectors, played the more significant role. While the simplicity of the AAMP-FV certainly made the initial development of the proofs and specifications easier, the AAMP5 and AAMP-FV specifications are of roughly comparable complexity. In fact, AAMP-FV macro and micro architecture specifications are actually larger than the corresponding AAMP5 specifications. This is because 1) much of the complexity of the AAMP5 was avoided by creating property oriented specifications of many components that abstracted away from internal details not required to prove the correctness of the microcode and 2) the AAMP-FV is specified in a different style that emphasizes clarity and the use of extensive comments.

On the AAMP5 project, most of the proofs of correctness were done by SRI, with approximately 800 hours spent verifying 11 instructions. A key goal of the AAMP-FV project was to ensure that Collins became experienced in using the PVS prover. For some classes of instruction, work on the AAMP5 provided sufficient experience for Collins to complete the proofs. For other classes, SRI developed the initial proofs and Collins completed the proofs for the remaining instructions in the class. SRI also focused on proofs that were common to all instructions, such as verifying the logic for stack adjustment.

As discussed in Chapter 6, the correctness proofs break down naturally into three parts, the *micro correctness* proofs that establish the correctness of the microcode at the register-transfer level, the proofs of common microcode such as that for adjusting the stack cache, and the *macro lift* proofs that show that the change at the register-transfer level implements the correct behavior at the instruction set level.

Table 3.1: Level of Effort

	Performed	Start	Stop	Hours
Project Management				
Planning	Collins	Oct 94	Oct 96	83
Weekly Meetings	Collins	Oct 94	Oct 96	172
Project Support				
Training	Collins	Mar 95	Aug 96	212
Configuration Management	Collins	Oct 94	Oct 96	26
Tool Support	Collins	Oct 94	Oct 96	48
Specification of the Macroarchitecture (3,693 Lines of PVS)				
Develop Macro Specification	Collins	Feb 95	Jun 95	130
Specification of the Microarchitecture (3,496 Lines of PVS)				
Develop Micro Specification	Collins	Nov 94	Feb 95	90
Translate Microcode	Collins	Feb 95	May 96	77
Proofs of Correctness - Standard Instructions				
Micro Correctness	Collins	Mar 95	Oct 95	477
	SRI			520
Macro Lifts	Collins	Aug 95	Mar 96	238
	SRI			40
Common to All Instructions	SRI			240
Clean-Up	Collins	Aug 96	Aug 96	69
Proofs of Correctness - Complex Instructions				
Micro Correctness	Collins	Oct 95	Jul 96	385
	SRI			
Macro Lifts	Collins	Apr 96	Aug 96	38

A summary of the AAMP-FV instructions and their proof status is shown in Table 3.2. Fifty-four of the 80 AAMP-FV instructions were verified. As can be seen in Table 3.1, 997 hours were spent on the micro correctness proofs and 278 hours were spent on the macro lift proofs for these instructions. The proof of these instructions were all similar, and after the first one in a class was completed, could be done in a few hours for each instruction. In many ways, this work typified an engineering process rather than a exploratory research program.

An additional 240 hours were spent verifying microcode common to all instructions, such as the stack adjust logic. While many of the techniques developed for verification of the simple AAMP-FV instructions could be applied here, this work was quite exploratory. Fortunately, it only had to be done once for this project.

During the course of the project, a number of proofs were broken as the specifications were changed to facilitate completion of other proofs. Rather than fixing these proofs immediately, they were left until all changes were completed and then fixed at the end of the project. Sixty-nine hours was spent on this activity.

If training is omitted, 2,074 hours were spent in direct verification of these 54 instructions.

Table 3.2: Proofs Completed

Instruction Class			**Proof**	
			Completed	**To Be Done**
Stack Management		5	DUP, DUPD, EXCH, EXCHD, POP	
Literal Data		5	LIT4, LIT16, LIT24, LIT32	LIT8
Reference Data		11	REF24, REF8, REFA, REFD24, REFD8, REFDA, REFDL, REFDL4, REFL, REFL4, REFMSK	
Assign Data		12	ASN24, ASN8, ASNA, ASND24, ASND8, ASNDA, ASNDL, ASNDL4, ASNL, ASNL4, ASNMSK	ASNBIT
Mutual Exclusion		1		SWAP
Operand Location		1		LOCL
Logical		4	AND, NOT, OR, XOR	
Arithmetic	Integer	10	ABS, ABSD, ADD, ADDD, SUB, SUBD	IDIV, IDIVD, IMPY, IMPYD
	Fractional	6		FDIV, FDIVD, FMPY, FMPYD, FMPYE, X5
Relational		4	EQ, EQD, GR, GRD	
Type Conversion		2	EXTS	TRUNC
Shift		4		SHL, SHLD, SHR, SHRD
Control	Branch	7	SKIP, SKIP8, SKIPF, SKIPF8, SKIPT, SKIPT8, JUMP24	
	Call	1		CALL
	Exception	3		CKINTS, CLRINT
	Return	1		RETURN
Context Switch		4		TRAP, USER
Miscellaneous		4	NOP	VSN

This gives us an average rate of about 38 hours per instruction, almost an order of magnitude reduction from the costs observed on the AAMP5 project. We believe the experience gained on the AAMP-FV project would allow us to cut this cost in half on a similar project.

However, the nature of the project changed qualitatively as the proofs progressed to instructions with more complex microcode, such as the multiply, divide, shift, call, and return instructions. Most of the AAMP-FV instructions have microcode simple enough that it can be verified directly through symbolic execution with PVS (see Section 6.2.1). This technique broke down on the more complex instructions because the expressions generated through symbolic execution grew too large to manage with PVS. As a result, the project entered into another exploratory phase as new techniques, such as proof by induction over the number of microcycles, were developed (Section 6.2.2). While very valuable, this phase was characterized by a high learning curve and very high costs per instruction. As with the AAMP5, we believe this cost could be dramatically reduced on subsequent projects.

Currently, the micro correctness and macro lift proofs are largely completed for the 54 instructions listed in Table 3.2. Of the complex instructions, Collins has completed the micro correctness

proofs of the CALL instruction and SRI has completed the micro correctness proofs of the IMPY and SHR instructions. Some supporting theorems that these proofs depend, such as lemmas involving bit-vector expressions, have not been completed. Completion of these proofs has been deferred because they are expected to be routine.[1]

[1]In a later phase of the project, SRI completed the proofs of these supporting lemmas, as well as the proofs of some of the more complex instructions such as CALL and IMPY. These proofs were ran top-to-bottom to ensure no lemma was left unproved in the proof chain. The axioms in the new specification have not been validated by Collins, but the proofs have been installed and executed by them. SRI also explored in this later phase ways to automate the proofs and make them more efficient. This work is documented in [33].

Chapter 4

The Macroarchitecture: The Programmer's View of the AAMP-FV

The AAMP-FV macroarchitecture is precisely the view of the AAMP-FV that an application programmer must understand to write assembly code. This section describes the AAMP-FV macroarchitecture informally, then describes how a formal model of the macroarchitecture was defined in PVS.

4.1 Overview of the AAMP-FV Macroarchitecture

Important features of the AAMP-FV macroarchitecture include its organization of memory, the process stack, stack cache, internal registers, instruction set, and support for multi-tasking and error handling. These are discussed in the following sections.

4.1.1 Organization of Memory

The AAMP-FV supports up to four separate address spaces, where each address space consists of 16M 16-bit words. Unlike the AAMP5, an AAMP-FV memory space is not segmented and is addressed via a single 24-bit address. Each address space is characterized as being either *code* or *data* memory and *user* or *executive* memory.

In a specific product, the system designers may choose whether to use the code/data and user/exec lines on the processor to associate separate physical memory with each address space. If they choose to use both lines, a memory access references one of four physical memory banks, one for each address space. If they choose to use neither line, all memory accesses reference the same bank regardless of the code/data and user/exec lines, folding the four address spaces onto the same physical memory. It is also possible to use only the code/data or the user/exec line when accessing memory, making four memory configurations possible.

4.1.2 Process Stack

As with all members of the AAMP family, the process stack is central to the AAMP-FV macroarchitecture, implementing in hardware many of the features needed to support high-level block

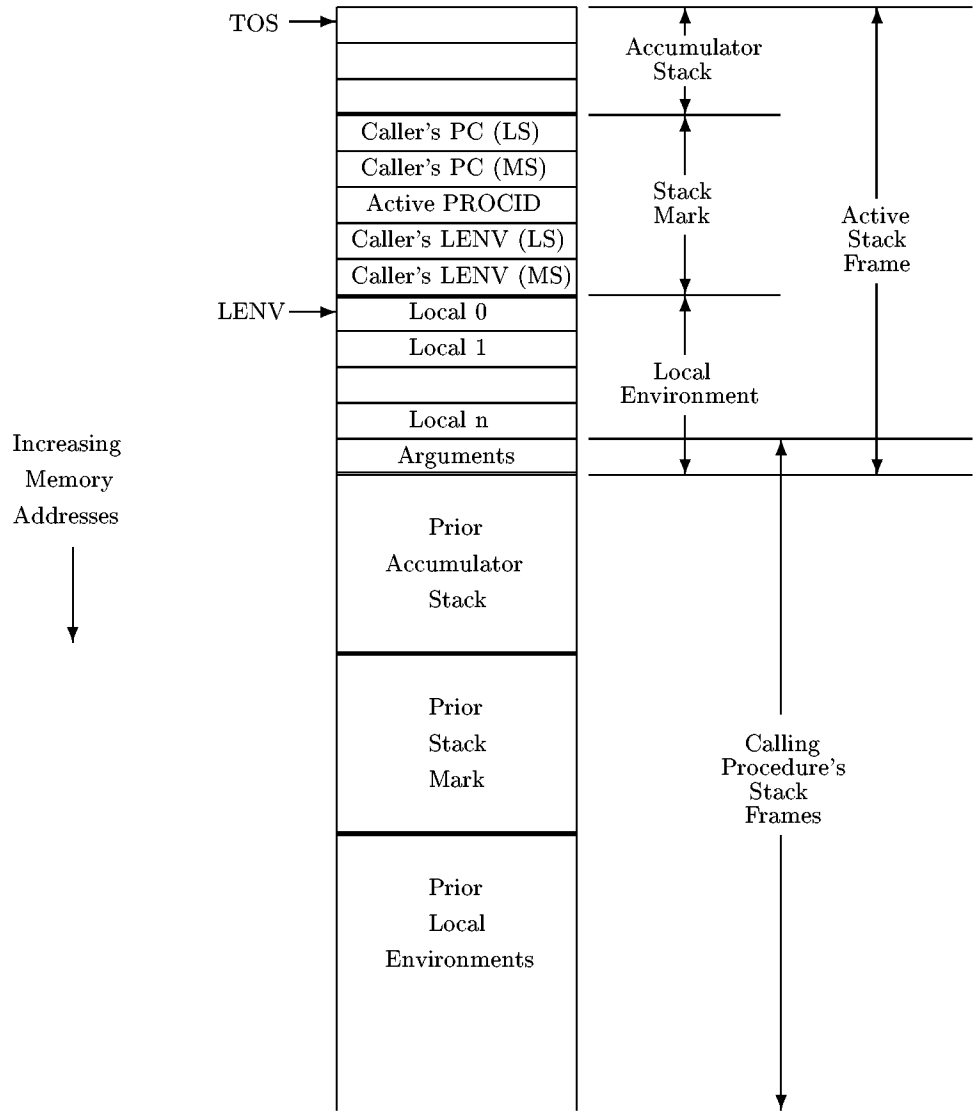


Figure 4.1: The Process Stack

structured languages and multi-tasking [3]. Each task maintains a single process stack, illustrated in Figure 4.1.

At the top of the process stack is the *accumulator stack* used for manipulation of instruction operands and pointers. Directly below the accumulator stack is the *stack mark* of the current procedure. The stack contains the information needed to restore the calling procedure upon return from the current procedure (Caller's PC and LENV, most significant and least significant words), to access local variables within the calling procedure (Caller's LENV), and to locate the current procedure's header and executable code (Active PROCID).

Below the stack mark is the current procedure's *local environment* consisting of its local variables and any parameters passed from the calling procedure. A procedure's local environment, stack

mark, and accumulator stack form a stack frame. Beneath the current procedure's stack frame is the frame of its calling procedure, and so on. Note that the stack grows downward towards decreasing memory addresses.

4.1.3 Stack Cache

As a stack machine, the AAMP-FV performs all data computations and manipulations on operands that have been pushed onto the top of the process stack. To improve efficiency, the top few words of the stack are actually maintained in internal registers referred to as the *stack cache*. Consistency between the stack cache registers and external memory is maintained through *stack adjustments* that read additional operands into the registers or write operands out to memory prior to the execution of each instruction. Studies have shown that stack adjustments are not required in about 95 percent of the instructions executed in a typical embedded application. This encachement technique is an essential performance feature of the AAMP-FV.

Ideally, the stack cache would be invisible to the application programmer. However, in the interest of efficiency, its presence is made visible to the application programmer in two ways. First, the AAMP-FV does not check memory accesses to determine if the word being referenced lies in the vicinity of the stack cache. As a result, REF (reference) and ASN (assign) instructions that address this region will obtain or modify the actual values stored in memory rather than those held in the stack cache. In practice, this does not pose a problem since well behaved applications do not directly read or write to memory used to implement the accumulator stack. Since applications seldom write assembly code for the AAMP-FV (recall that it is designed for use with high order, block structured languages such as Ada), this is primarily a concern for the compiler writers.

Second, as the process stack shrinks the words of memory uncovered by the stack may or may not contain the values the application programmer expects, depending on whether a stack cache adjustment had reconciled the contents of the stack cache with physical memory. Again, this is not a problem in practice as application programs should not directly access the area of memory reserved for the accumulator stack. However, a formal specification of the AAMP-FV requires that this behavior be captured in the macro-architecture specification as discussed in Section 4.2.4.

4.1.4 Internal Registers

The AAMP-FV maintains several internal registers that are visible to the application programmer in that they determine how the processor executes each new instruction. The TOS (top of stack) register points to the topmost word in the process stack. The LENV (local environment) register points to the local environment of the current procedure and is used in addressing local variables. The PAGEREG (page register) register maintains the base address used in paged memory addressing mode. TOS, LENV, and PAGEREG are implemented as 32-bit registers, although only the bottom 24 bits are used. The PC (program counter) contains the *byte* address of the next instruction to be executed and is actually implemented as a 24-bit address register.

In addition, the AAMP-FV maintains the UM (user mode) bit and MASK and INTREG registers. The UM bit is set high while the processor is in user mode and low while in executive mode. This value is brought out of the processor via the user/exec line and can be used to distinguish user and executive mode memory references as discussed in Section 4.1.1. Finally, the INTREG register holds the status of 8 prioritized interrupts, and the MASK register is used by the application pro-

grammer to mask (inhibit) these interrupts. The two highest priority interrupts are non-maskable, i.e., they cannot be inhibited by the application programmer.

4.1.5 Instruction Set and Data Types

The AAMP-FV instruction set consists of 80 instructions and is CISC-like, closely resembling the intermediate output of most compilers. Instructions area all 8 bits long, yielding high throughput and code density. The instruction set supports 16-bit and 32-bit integers, 16-bit and 32-bit fractional number, and 16-bit logical variables.

The instruction set can be divided into several classes, as shown in Table 3.2 on page 11. Of the 80 AAMP-FV instructions, 23 are Reference or Assign instructions that move data between the top of the process stack and data memory. The Logical, Arithmetic, Relational, Type Conversion, and Shift instructions, which perform a prescribed operation on the top few elements of the process stack and push the result back onto the stack, account for an 30 instructions. An additional 12 instructions deal with program control, such as branch, call, return, and interrupt handling. The remaining instructions duplicate or move operands on the top of the stack, push literal data onto the stack, support mutual exclusion and operand location, or perform miscellaneous functions such as NOP (no operation).

4.1.6 Multi-Tasking and Error Handling

The AAMP-FV stack architecture is designed for real-time multi-tasking applications where the processor is time-shared among two or more concurrent tasks. Each task maintains its own process stack in memory, along with a single stack for the executive. This provides an efficient means to suspend and resume each task since only the contents of internal registers need to be saved and restored. If the user/exec line is used to partition memory addresses, the executive stack will reside in a separate memory space from the user stacks.

At system initialization or following a system reset, the processor is placed into executive mode and begins execution of the executive procedure, reading the address of the executive procedure and stack from known locations in memory. The executive selects the next user task to be activated or resumed, places the PSD (Process State Descriptor) of the user task on the top of the executive stack, and executes a USER instruction to initiate a context switch to user mode. This consists of storing the processor's registers in the executive PSD, loading its registers from the the user PSD, and setting the UM bit high. The processor then starts execution of the user task, continuing until an interrupt or trap occurs.

Interrupts are asynchronous hardware inputs to the processor, while traps are generated by software, usually through execution of a TRAP instruction. The occurrence of a hardware interrupt or a TRAP while in user mode initiates a context switch by the processor back to executive mode, reversing the earlier context switch. First the UM bit is set low, the processor's registers are stored in the user PSD and loaded from the executive PSD, and the interrupt or trap number is pushed on the executive stack. The processor then resumes execution of the executive process.

Control is passed back and forth between the executive and user tasks following this protocol. Unrecoverable errors encountered while in user mode, such as execution of an illegal instruction, can also cause control to be transferred back to the executive. For example, execution of the USER instruction while in user mode is treated as an illegal instruction and transfers control back to the executive. Unrecoverable errors encountered while in executive mode place the processor in

an error state in which it idles pending reset. These include execution of an illegal instruction or TRAP instruction while in executive mode.

Up to eight prioritized interrupts can be stored in the INTREG register. Of these, the lower six can be masked by the application by setting bits in the MASK register, inhibiting transfer back to the executive even when the corresponding interrupt is pending. The two highest priority interrupts cannot be masked by an application task. An interrupt causing a transfer to executive mode is automatically cleared by that context switch. The AAMP-FV also provides a RST (reset) line that can be used to cause a processor reset. This can be viewed as the highest level interrupt.

Interrupts are serviced only at instruction boundaries and only while in user mode. Interrupts that occur while in executive mode (other than a system reset) are held pending in the INTREG register until control is transferred back to a user task, at which time they initiate a context switch back to executive mode. Of course, the status of the INTREG register can be interrogated by the executive and the executive could be designed to avoid switching to user mode while interrupts are pending if desired.

Exceptions, such as arithmetic overflow, do not cause an automatic switch to executive mode as in the AAMP5. Instead, exceptions raise the OVR (overflow) line from the processor. This signal can be used on the external circuit board to raise one of the eight hardware interrupts, causing the exception to be handled via the normal interrupt handling mechanism.

4.2 Formal Specification of the Macroarchitecture

The macroarchitecture specification formalizes the assembly-level programmer's view of the AAMP-FV and its instruction set. The PVS specification of the AMMP-FV models the processor as a state machine. The state of the macro machine includes external memory and the internal state that affects its observable behavior, such as the internal registers defining the process stack. The next state function specifies the effect of executing the current instruction pointed to by the program counter. An overview of the *import chain* for the macroarchitecture specification is shown in Figure 4.2.

In PVS, a theory gains access to another theory's definitions and axioms by *importing* that

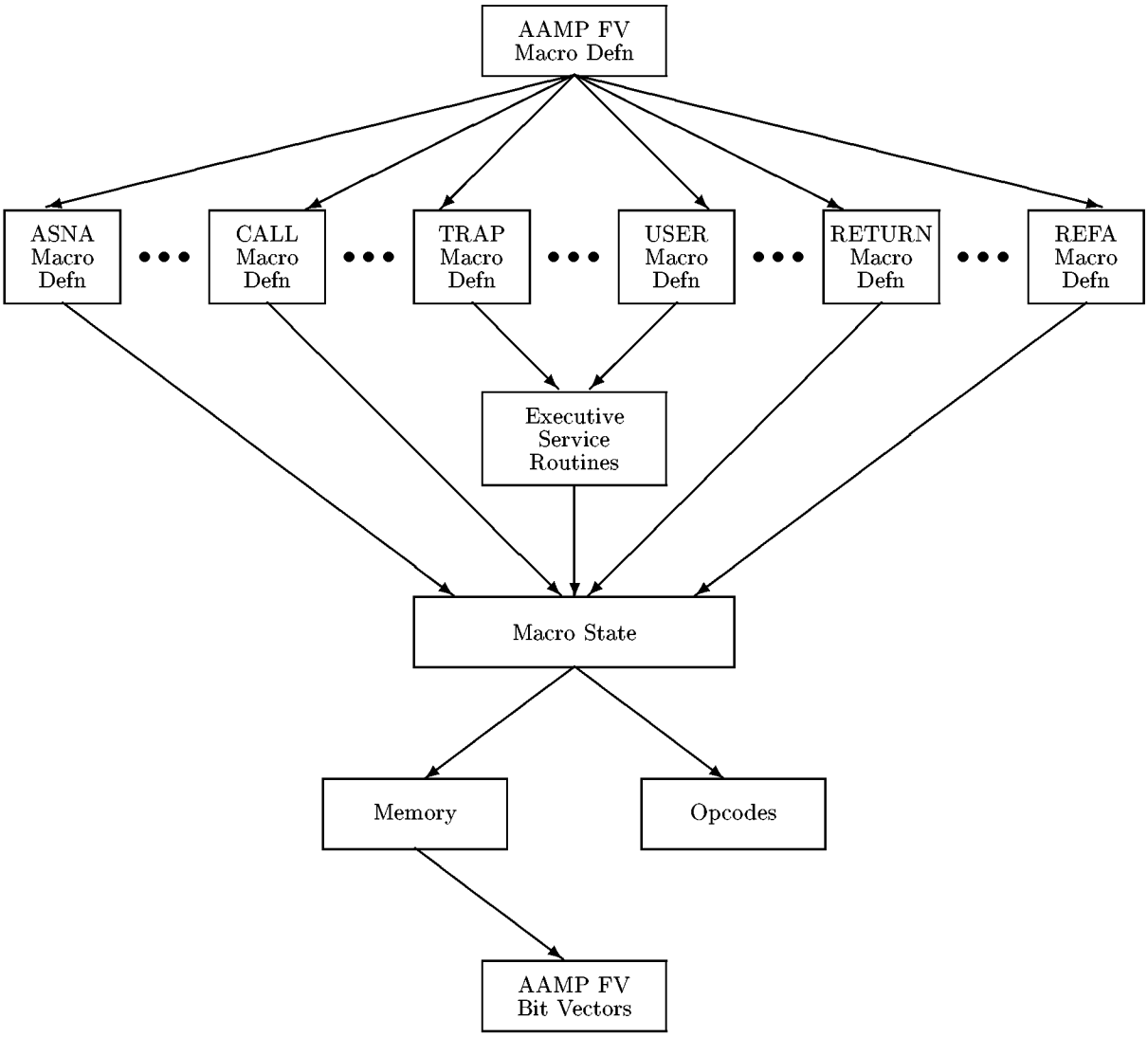


Figure 4.2: Macroarchitecture Specification Hierarchy

theory. Each box in the figure represents a theory in the specification. Importation of a theory is depicted by an arrow from the importing theory to the imported theory.

At the topmost level is the `AAMP_FV_macro_defn` theory. This theory simply imports the definition of each instruction, each of which is defined in its own theory. The majority of these are defined directly in terms of the change they cause in the macro architecture state, defined in the `macro_state` theory. A few of the more complex instructions, such as `TRAP` and `USER` that involve a context switch, import additional definitions such as the `executive_service_routines`. Since the macroarchitecture state includes external memory, the `macro_state` theory imports the definition of memory, which in turn imports the definition of AAMP-FV specific bit vectors. Although it is not shown in Figure 4.2, the `AAMPFV_bit_vectors` imports the bit vectors library in which the detailed properties and operations of bit vectors (i.e., sequences of bits) are defined. The `macro_state` theory also imports the definition of the AAMP-FV opcodes. The following sections discuss these theories in greater detail.

4.2.1 Bit Vectors

The `AAMPFV_bit_vectors` theory, shown in Figure 4.3, defines some of the most common bit vector types and constants used in the specification of the AAMP-FV. Another important role of this

```
AAMPFV_bit_vectors: THEORY
BEGIN
  IMPORTING bv_top

  %-------------------------------------------------------------------------
  % Common bit vectors
  %-------------------------------------------------------------------------
  byte          : TYPE = bvec[8]
  word          : TYPE = bvec[16]
  address       : TYPE = bvec[24]
  register      : TYPE = bvec[32]

  %-------------------------------------------------------------------------
  % Common conversions
  %-------------------------------------------------------------------------
  r2a(r : register) : address  = r^(23,0)
  a2r(a : address)  : register = fill[8](0) o a

  %-------------------------------------------------------------------------
  % Common constants
  %-------------------------------------------------------------------------
  Hx00          : bvec[8]              = nat2bv(0)
   ...                    ...                    ...
  Hx80000000    : bvec[32]             = int2bv(-exp2(31))

END AAMPFV_bit_vectors
```

Figure 4.3: PVS Specification of AAMP-FV Bit Vectors

theory is to provide a single point for importing the bit vectors library, via theory `bv_top`, in which the representation and operations of the bit vectors are defined. Operations such as concatenation (`o`), extraction (`^`), addition (`+`), and subtraction (`-`) of bit vectors are defined in this library. AAMP-FV specific functions, such as `r2a` that converts a 32-bit register to a 24-bit address and a2r that converts a 24-bit address to a 32-bit register, are defined in terms of the more fundamental extraction and concatenation operations. Details of the bit vectors and their operations can be found in [10].

4.2.2 Memory

As discussed in Section 4.1.1, AAMP-FV memory can be configured in four different ways by the system designer. The PVS specification of AAMP-FV memory, shown in Figure 4.4 defines all four possible configurations.

A *memory space* is defined as a function mapping 24-bit addresses into 16-bit words of memory. Memory itself is defined as four memory spaces, where each memory space is indexed by two boolean values representing the value of the code/data and user/exec lines. All accesses to memory are then written in terms of two functions, `read` and `write`. `Read` takes an address, specified as a value of the code/data line, user/exec line, and a memory address, and returns a word of memory. Write takes a similar address triple and a word and updates memory at the specified address. Whether the memory interface unit uses the code/data or user/exec lines to partition memory is encoded in two boolean constants, `separate_code_data_memory_spaces` and `separate_user_exec_memory_spaces`. If both of these constants are true, then both the code/data and user/exec arguments to `read` and `write` are used to direct access to the appropriate memory space. If both are false, then the code/data and user/exec arguments are ignored and only the `(false,false)` partition is ever accessed. Two additional configurations can be modeled by setting one of the constants true and the other false.

The actual values of `separate_code_data_memory_spaces` and `separate_user_exec_memory_spaces` are left as unspecified constants. In this way, the correctness proofs are valid regardless of which memory configuration is chosen by the system architect. This typically manifests itself as one or two additional proof branchs in the macro-lift proofs (Section 6.4).

4.2.3 Macroarchitecture State

The *macroarchitecture state* defines the portion of the AAMP-FV state seen by the application programmer. Its specification in PVS is given in theory `macro_state` shown in Figure 4.5. The macroarchitecture state consists of an instantiation of memory (defined in Figure 4.4), the PAGEREG, TOS, PC, LENV, MASK, and INTREG registers, and the UM flag (described in Section 4.1.4).

Also included in the `macro_state` theory are a number of auxilliary functions closely related to the macroarchitecture state. For example, `fetch` returns a byte of code memory located at a specific *byte* address. Note that it right shifts the byte address to form a word address, uses the `read` function to retrieve that word of code memory, and then uses the low-order bit of the address to select the appropriate byte of the word retrieved. This function is typically used to reference bytes of immediate data using the PC as an operand. Defining it once here avoids repeating this operation throughout the macroarchitecture specification. `Top` is another useful function; it returns the ith word from the top of the process stack. The `next_macro_state` function is discussed in the next section.

```
memory: THEORY

BEGIN

  IMPORTING AAMPFV_bit_vectors

  %--------------------------------------------------------------------------
  % A memory space is a function from a 24-bit address to words of memory.
  % Memory consists of four memory spaces indexed by the values of the
  % code/data and user/exec lines.
  %--------------------------------------------------------------------------
  memory_space  : TYPE = [address -> word]
  memory        : TYPE = [ [bool,bool] -> memory_space ]

  %--------------------------------------------------------------------------
  % The two following boolean constants determine which of the four possible
  % memory models is used. They are deliberately left unspecified.
  %--------------------------------------------------------------------------
  separate_code_data_memory_spaces: bool
  separate_user_exec_memory_spaces: bool

  %--------------------------------------------------------------------------
  % Write updates a word in one of the four memory spaces.
  %--------------------------------------------------------------------------
  write(cd: bool, ue: bool, a: address, w: word, m: memory) : memory =
   IF separate_code_data_memory_spaces
   THEN IF separate_user_exec_memory_spaces
           THEN m WITH [(cd,ue)(a)       := w]
           ELSE m WITH [(cd,false)(a)    := w]
        ENDIF
   ELSE IF separate_user_exec_memory_spaces
           THEN m WITH [(false,ue)(a)    := w]
           ELSE m WITH [(false,false)(a) := w]
        ENDIF
   ENDIF

  %--------------------------------------------------------------------------
  % Read retrieves a word from one of the four memory spaces.
  %--------------------------------------------------------------------------
  read (cd: bool, ue: bool, a: address, m: memory) : word =
   IF separate_code_data_memory_spaces
   THEN IF separate_user_exec_memory_spaces
           THEN m(cd,ue)(a)
           ELSE m(cd,false)(a)
        ENDIF
   ELSE IF separate_user_exec_memory_spaces
           THEN m(false,ue)(a)
           ELSE m(false,false)(a)
        ENDIF
   ENDIF

 END memory
```

Figure 4.4: PVS Specification of AAMP-FV Memory

```
macro_state: THEORY

BEGIN

  IMPORTING memory, opcodes

  %-----------------------------------------------------------------------------
  % AAMP-FV macro state
  %-----------------------------------------------------------------------------
  macro_state: TYPE = [# mem     : memory,       % Memory
                         pagereg : register,     % Base address for paged data
                         tos     : register,     % The top of the process stack
                         pc      : address,      % Program counter
                         lenv    : register,     % Local environment
                         um      : bool,         % User/executive mode
                         mask    : byte,         % Interrupt mask
                         intreg  : byte #]       % Interrupt state.

  %-----------------------------------------------------------------------------
  % Fetches the byte of code memory at byte address a
  %-----------------------------------------------------------------------------
  fetch (ue: bool, a:address, m:memory): byte =
    LET w = read(code, ue, fill[1](0) o a^(23,1), m)
    IN IF ishigh(a^0) THEN w^(15,8) ELSE w^(7,0) ENDIF

  %-----------------------------------------------------------------------------
  % Returns the ith word from the top of the process stack.
  %-----------------------------------------------------------------------------
  top(st:macro_state, i:nat):word = read(data, um(st), r2a(tos(st))+i, mem(st))
   ...                      ...                      ...
  %-----------------------------------------------------------------------------
  % Defines the next state of the macro machine.
  %-----------------------------------------------------------------------------
  next_macro_state: [macro_state -> macro_state]

END macro_state
```

Figure 4.5: PVS Specification of AAMP-FV Macroarchitecture State

4.2.4 Next Macro State Function

Each AAMP-FV instruction is specified in PVS as a state transition function, `next_macro_state`, over the macroarchitecture state defined in Figure 4.5. For convenience, the specification of each instruction is placed in a separate theory. For example, the specification of the REFA instruction is given in theory `REFA_macro_defn` shown in Figure 4.6. The REFA instruction pops two words off the top of the process stack, concatenates them together and uses the lower 24 bits as the address `WA` of a word of memory `XS` to be read and placed on the top of the stack. The axiom in Figure 4.6 states that if the current opcode is `REFA` and the word address `WA` and program counter do not lie in the region of the stack cache, the new macro state is the current macro state `st` with an unspecified

```
REFA_macro_defn [(IMPORTING AAMPFV_bit_vectors)
                 unspecified: [nat -> word]] : THEORY
BEGIN

  IMPORTING macro_state

  st: VAR macro_state

  %-----------------------------------------------------------------------
  % The REFA instruction uses the double word that is at the top of the
  % stack as the absolute address for a single precision read, pushing the
  % word read on the stack.
  % -----------------------------------------------------------------------
   REFA: AXIOM
     (current_opcode(st) = REFA &
        not_stack_cache_address(st)(WA) &
          pc_not_in_cache_region(st) =>
        next_macro_state(st) =
          (st WITH [(mem) := write(data, um(st), r2a(newtos-1), unspecified(0),
                             write(data, um(st), r2a(newtos),   XS, mem(st))),
                    (pc)  := pc(st)  + 1,
                    (tos) := newtos]
          ) WHERE XS     = read(data, um(st), WA, mem(st)),
                  newtos = tos(st) + 1
     ) WHERE WA  = (top(st,1) o top(st,0))^(23,0)

END REFA_macro_defn
```

Figure 4.6: PVS Specification of REFA instruction

word written at the old top of stack (`newtos - 1` [1]), the word `XS` located at memory location `WA` written at the new top of stack location (`newtos`), the program counter `pc` incremented by one, and the top of stack pointer `tos` set to its new value (`tos(st) + 1`).

The presence of the stack cache is visible in the REFA instruction in both of the ways mentioned in Section 4.1.3. First, as indicated by the `not_stack_cache_address` and `pc_not_in_stack_cache_region` predicates in the antecedent, the behavior of the instruction is not specified if the word being referenced or the program counter lie in the vicinity of the stack cache. Second, the word of memory uncovered by the shrinking of the process stack has been set to an unspecified word. Both the predicates `not_stack_cache_address` and `pc_not_in_stack_cache_region` and the function `unspecified` are left uninterpreted in the macroarchitecture specification. Thus the application programmer knows only that he can make no assumptions about the REFA instruction if the word being referenced or the program counter lies in the vicinity of the stack cache (conservatively taken to be at least the top 6 words of the stack). In similar fashion, he knows only that he will find some word, `unspecified(0)`, in memory at the location previously occupied by the top of the stack, but cannot make any assumptions about the value of that word.

[1] Recall that the stack grows downward and two address words were popped prior to pushing the referenced word onto the stack.

At the microarchitecture level, the values of these functions are defined and are used to complete the proofs of correctness. The `not_stack_cache` and `pc_not_in_stack_cache_region` functions are precisely defined in terms of internal registers not visible to the application programmer and the function `unspecified` is instantiated with the actual value that will be found at that location in order to take the proof to completion.

Chapter 5

The Microarchitecture: The Register Transfer View of the AAMP-FV

5.1 Overview of the AAMP-FV Microarchitecture

The microarchitecture specification describes the AAMP-FV at the register-transfer level, i.e., it specifies the effect of an arbitrary microinstruction on the movement of data between the registers and other components of the AAMP-FV. An overview of the microarchitecture is shown in Figure 5.1. The AAMP-FV microarchitecture can be divided into three main parts: the data path, the microcontroller, and the bus interface unit.

5.1.1 The Data Path

The data path provides the data manipulation and processing functions required to execute the AAMP-FV instruction set. It consists of a 16-word multi-port register file, a 32-bit arithmetic logic unit (ALU), shift logic, data and address interface, address incrementors, and instruction register and parsing logic. To make address computations fast and to support double-precision arithmetic instructions efficiently, internal data paths are mostly 32-bits wide.

The register file is a key element of the microarchitecture. Its multiport design is important in achieving the parallelism needed for high execution speed and compact microcode. The register file contains the program counter (PC), Q register used in shift operations, interrupt mask (MASK), top of stack (TOS), local environment (LENV), page register (PAGE), four scratch pad registers (R0-R3), and the stack cache registers (STK0-STK5). Any of these registers can be output on the 32-bit A and B ports by providing the appropriate values to the A and B address inputs. Separate ports are provided for shifting the Q register, providing a mask to the interrupt controller, and for external address generation.

Register file entries STK0 through STK5 comprise the stack cache and can contain up to six 16-bit operands from the top of the process stack, with the remainder of the stack residing in external memory. Since these registers are addressed and output in pairs, 32-bit operands can be processed just as efficiently as 16-bit operands.

Single position shifts to the left or right are provided. Eight shift linkages provide efficient implementation of integer and fractional multiplication and division as well as shift instructions.

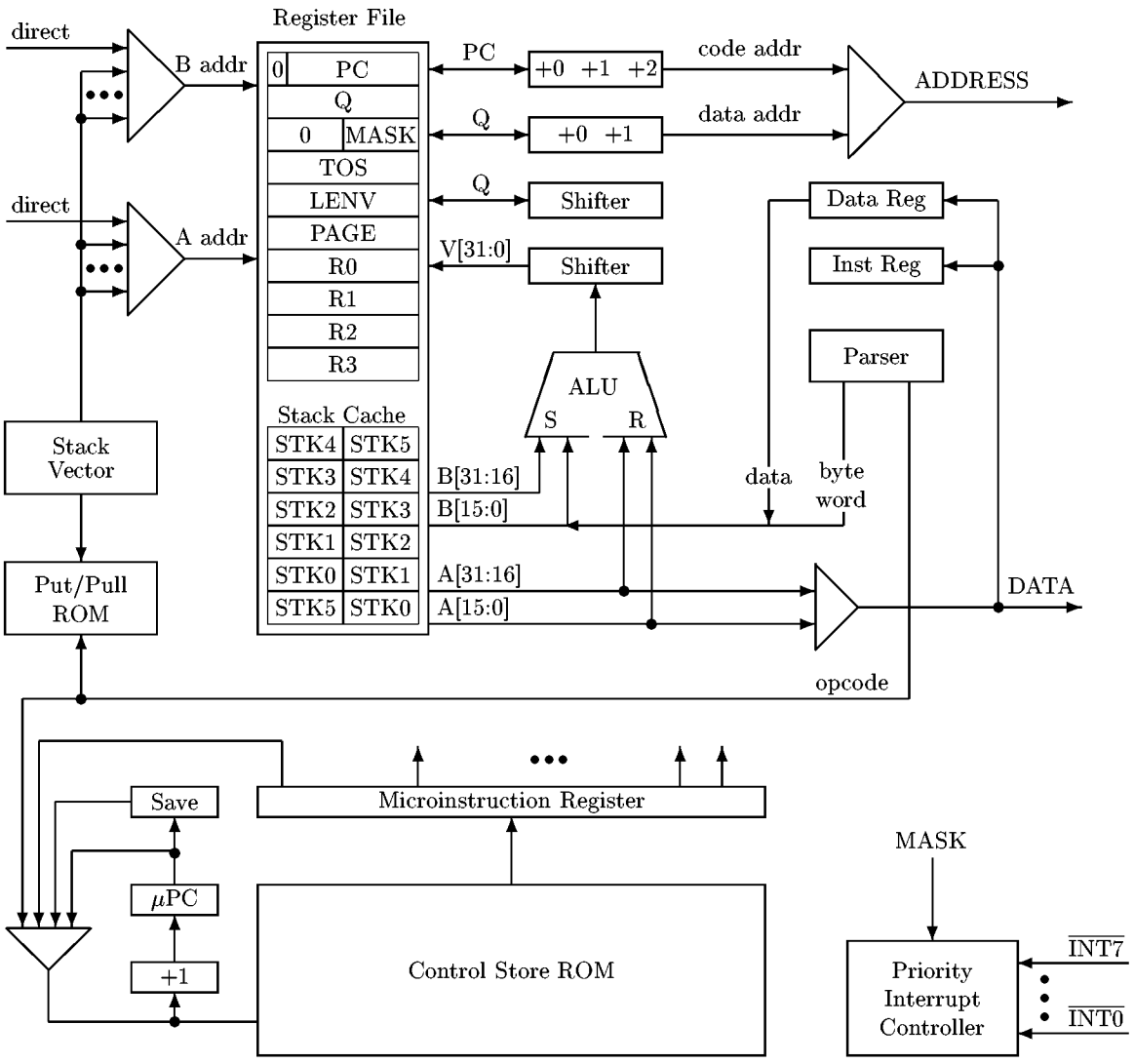


Figure 5.1: The AAMP-FV Microarchitecture

The ALU provides addition, subtraction, and logical operations on 32-bit bit-vectors. It also provides indications of sign, all-zero, carry, and 16-bit and 32-bit overflow. The R and S inputs to the ALU are fed from multiplexing logic that allows the inputs to be drawn from a number of different sources, including variations of the register file outputs, 16-bit data read from memory, immediate byte and word fields from the instruction stream, and microconstants.

5.1.2 The Microcontroller

The microcontroller consists of the microcode ROM, the logic to sequence and execute microinstructions, and the stack adjust and interrupt logic.

The AAMP-FV is a microprogrammed machine wherein control is via a stored program in ROM rather than discrete logic. This results in two levels of stored program in a system: one at the micro level using microinstructions in the control-store ROM, and the other at the macro level using machine language instructions stored in external code memory. In essence, each machine language instruction is interpreted as a pointer to a sequence of microinstructions to be executed. Each microinstruction causes one or more elemental operations to occur in the machine, such as enabling a register to be loaded or selecting an ALU function to perform.

Instruction bytes are fetched from code memory two at a time and stored in the instruction register. Execution begins with the translation of the opcode byte into a starting microprogram address. The microinstruction at this location is then loaded into the microinstruction register, the outputs of which configure the data paths and determine which operations are to occur during the current microcycle.

While the current microinstruction executes, the microsequencer determines the address of the next microinstruction to be executed. This can be the address of the current microinstruction incremented by one, a jump address contained in the current microinstruction, a saved register loaded from the microprogram counter to establish return linkage from a called microsubroutine, or fixed addresses for initialization, interrupt servicing, and stack cache adjustments. In some cases, the next microinstruction is conditional on the the state of selected status line.

Completion of the microprogram associated with the current machine instruction repeats this cycle, causing the address associated with the next opcode to be loaded into the microsequencer. An exception to this occurs when an interrupt is pending.

Interrupts are processed only at completion of the current machine instruction. While the current machine instruction executes, up to eight prioritized interrupts can be captured in an 8-bit register. Once the current machine instruction completes, the microcontroller checks if an unmasked interrupt is pending. If so, it selects the microaddress of the highest priority interrupt service routine for execution. Each interrupt is reset when it is processed, and once all interrupts are processed execution resumes with the next machine instruction.

The microsequence for each instruction assumes that sufficient operands are present in the stack cache for the instruction to execute and that sufficient room is present in the stack cache to hold the outcome of the instruction. The current status of the stack cache is maintained in the stack vector register. At the start of each machine instruction, the microcontroller feeds the opcode and the stack vector into the Put/Pull ROM, which determines if a *stack adjustment* is necessary. If an adjustment is needed, the microcontroller enters one of two microsequences that either reads an operand from memory into the stack cache or writes a word from the stack cache into memory. Upon completion of this microsequence, the microcontroller restarts the machine instruction.

5.1.3 The Bus Interface Unit

The bus interface unit (BIU) contains the logic needed to move data between the AAMP-FV and main memory, including the functions of bus arbitration, address generation, and data parsing.

Data is read or written to memory using the lower 24 bits of the Q register to address memory. When writing, the operand is selected from either the high or low 16-bits from the register file's A port, simplifying the handling of both single and double precision writes. Data read from memory is passed to the ALU as an S source. The register file also supports separate loading of the high and low halves of 32-bit destinations, accommodating the efficient transfer of 32-bit operands into the processor.

Instructions are fetched from code memory into the instruction register two bytes at a time. These may consist of any combination of opcodes or immediate data. Opcodes are passed to the microcontroller to initiate instruction execution. Immediate data bytes are fed to the ALU as S-source operands. Since instructions are one byte in length, the 16-bit instruction register provides partial look-ahead. When it is time to fetch an instruction, conditional logic first checks to determine if the instruction is already present in the instruction register from the prior fetch.

The program counter is a 24-bit byte address. Since it is a byte address, must be shifted right one bit with a fill of zero to form a word address with the least significant bit of the original PC selecting between the high and low byte returned from memory. As a result, although the AAMP-FV has a 2^{24} word address space, programs can only reside in the lower half of the address space.

Several AAMP-FV instructions use immediate data embedded in the instruction stream. To maintain code density, immediate data is not required to be word aligned. To avoid manipulating this data in the ALU, the BIU parses and extracts it from the instruction stream.

The BIU communicates with memory via several signals. The two primary transaction control signals are the transaction request signal and the read/write line. The transaction request signal indicates when the AAMP-FV is ready to perform a memory transaction. The read/write line establishes whether the transaction is a read or write request. The exec/user line indicates the mode (executive or user) of the processor and the code/data line indicates if the transaction is a code (or immediate data) fetch or a data memory transaction. The exec/user line and the code/data line can be used by external memory to select different memory spaces. The address of the desired word within this memory space is indicated by a 24-bit address bus. Data is transferred over the 16-bit data bus.

5.2 Formal Specification of the Microarchitecture

The PVS microarchitecture specification is a formal description of the the AAMP-FV microarchitecture and microcode. Each major component of the microarchitecture, such as the ALU or the register file, is described in one or more PVS theories. These specifications, together with a variety of "glue" theories describing the data and control paths between the components, define the microarchitecture over which the microcode executes. Translation of the microcode into PVS results in a specification that defines how the microarchitecture state and memory are altered by the execution of each microinstruction.

The microarchitecture specification describes the AAMP-FV from the perspective of a microcode programmer and abstracts away some of the details of an actual hardware implementation.

For example, time is modeled in the microarchitecture using the natural numbers, where one unit of time corresponds to one microcycle, i.e., the execution of one micro instruction. However, in an actual implementation, each microcycle would consist of one or more clock cycles, or phases. Thus, even though a memory read or write may actually take an indeterminate but finite amount of time, to the microcode each memory access takes one cycle. By abstracting away from the physical clock, it is possible to provide a more concise definition of the microarchitecture and simplify the proofs.

Every flip-flop, wire, register, and bus in the AAMP-FV microarchitecture is defined as a signal of some type, where a signal is a function from time to a type such as a bit or a bitvector. Some signals represent elements of the current "state" of the microarchitecture such as registers and external memory. Other signals represent "connectors", such as bus lines, and are defined as a combinatorial function of the current state.

At least two styles are commonly used in the formal specifications of hardware. In the *functional* style, one defines the output signals of a component as functions of its input signals, letting the signal definitions implicitly specify the connectivity between the components. In the *predicative* style [18] commonly used in HOL [19], every hardware component is specified as a predicate relating the input and output signals of the component. A design is specified in the predicative style as a conjunction of the the component predicates, with signals on the internal wires used to connect the components hidden by existential quantification.

The functional style of specification has several advantages in PVS. In particular, proofs are able to exploit the automatic rewriting capabilities of PVS and tend to be more automatic than when the specification is written in a predicative style. However, there are some difficulties with using the functional style in PVS. The first is that a name cannot be used until it is defined, making it difficult to model the feedback found in sequential circuits. It is also difficult to define a hierarchy of circuit blocks. Hierarchy can be emulated using parameterized theories, where the theory parameters represent the circuit inputs, but no similar mechanism is available for defining the circuit outputs.

To address these issues, the AAMP-FV specification is based on the notion of a "backplane" rather than a hierarchy. The backplane consists of a theory `SIG` that defines the name and type (but not the functional behavior) of all the registers, flip-flops, and connecting signal in the AAMP-FV. In this way, signals can be referenced even though their functional behavior has not yet been defined.

Each major functional block is described in its own theory. To connect the functional blocks together, three special purpose parameterized theories are defined, `CONNECT`, `DFF`, and `DFFR`. `CONNECT` (Figure 5.2) has three parameters, a type and two signals of that type, and states axiomatically that the second signal is equivalent to the first. `DFF` has the same parameters as `CONNECT`, but states that the value of the second signal at time (t+1) is equivalent to the value of the first at time t. `DFF` thus defines a D-type flip flop, where the first signal is the D input and the second is the Q output. The `DFFR` theory defines a resettable D-type flip flop and adds parameters for a value to which the flip flop is reset and a reset signal.

In this way, the `SIG` theory serves as a "backplane", making the definitions of the signal available wherever they are used in the microarchitecture specification. `CONNECT` is used to define a direct connection between two signals, `DFF` is used to connect two signals through a flip-flop, and `DFFR` is used to connect two signals through a resettable flip-flop.

Figure 5.3 shows the definition of the `Next_PC` (next program counter) register. Immediately following its functional specification is the instantiation of theory `PC` that connects the `Next_PC`

```
CONNECT [(IMPORTING AAMP_FV_basics)

    sig_type : TYPE,
    A        : signal[sig_type],
    B        : signal[sig_type]

]:  THEORY

BEGIN

  t : VAR time;

  CONNECT: AXIOM B(t) = A(t)

END CONNECT
```

Figure 5.2: PVS Specification of CONNECT

```
    Next_PC(t) : bvec[24] = IF LPC(t) THEN
                              V(t)^(23,0)
                            ELSE
                              CASES PQ(MC(t)) OF
                                PC      : NxPC(t),
                                PCplus1 : NxPC(t),
                                PCplus2 : NxPC(t)
                                ELSE      PC(t)
                              ENDCASES
                            ENDIF

      PC: THEORY = DFFR[bvec[24],Next_PC,PC,bvec0[24],RST]
```

Figure 5.3: PVS Specification of the Next PC Register

register to the PC register through a flip-flop that can be cleared by the RST signal. The other names in this specification fragment, such as LPC, V, and MC are all signals named in SIG and defined in other parts of the microarchitecture specification.

Use of theories such as CONNECT, DFF, and DFFR reduced the complexity of the proof scripts by making it simple to load the connections as auto-rewrites. For example, to cause the PC-Next_PC connection to be treated by PVS as an automatic rewrite rule during a proof, it is only necessary to issue the prover command (auto-rewrite-theory "PC"). This makes the proof script more readable and manageable for the engineer performing the proof.

Specifying the microarchitecture with a "backplane" reflected the actual design of the AAMP-FV, making construction of the formal model straight forward. As pointed out in Section 3, specification of the microarchitecture was completed in approximately 90 man hours. The functional style also reduced the likelihood of introducing inconsistencies in the specification. Although the

DFF and CONNECT theories are axiomatic, they are used in a controlled manner that minimizes the potential for error.

The microarchitecture specification of the AAMP-FV is very similar to what an HDL specification of the processor would look like. In large part, this is due to the language features of PVS that map readily to similar HDL constructs. This has important benefits when working with engineers unfamiliar with formal specification languages, since they can read a PVS specification and understand what is being said without extensive training.

5.3 Formal Specification of the Microcode

The microcode for the AAMP-FV was translated by hand into PVS. An example for the REFA instruction is shown in Figure 5.4. While the PVS representation is not easily read (except by an AAMP-FV microcode programmer), it has the advantage that it was generated from the original microcode by a very straight forward process. In future projects, it would not be possible to build a translator that would generate the PVS representation automatically from the original microcode.

```
ucode_REFA : THEORY

BEGIN

  IMPORTING EP_basics
  IMPORTING Put_Pull_ROM_basics
  IMPORTING micro_macro_defns

%% Define the Entry Point

    REFAep : AXIOM  EP_OF(REFA) = EP_REFA;

%% Define the Entry Condition

    PP_REFA: AXIOM PP_ROM(REFA) = Pullif_OClt2    %% IF(SV>1)DO

%% Define the Microcode

    REFAinstrn: AXIOM uROM(EP_REFA) =

      FeqR(                                     %% F<-R
        ReqSVminus1_O(                          %% R<-VM1:V
          INC_PCandCFETCH(                      %% FCON+1
            JMP(REFS1, default_minstrn))))      %% => REFS1

            WITH [(DN)  := QgetsF,              %% Q<-F
                  (PP)  := POP2]                %% POP2
END ucode_REFA
```

Figure 5.4: PVS Specification of the REFA Microcode

Chapter 6

Formal Verification of the AAMP-FV

6.1 Overview

Section 2.6 discussed the kinds of theorems to be proved in verifying the microcode of a microprocessor. In this section we describe the specific theorems proved about the AAMP-FV. Sections 6.2, 6.3, and 6.4 describe in detail what has been proven about the AAMP-FV microcode using PVS.

6.1.1 Commutativity Theorems

Most of the effort spent proving the correctness of the AAMP-FV microcode focused on verification of the commutativity theorem (illustrated in Figure 6.1) relating the microarchitecture and macroarchitecture models described in Chapters 4 and 5. The proof of the microcode commutativity theorem divides naturally into three parts.

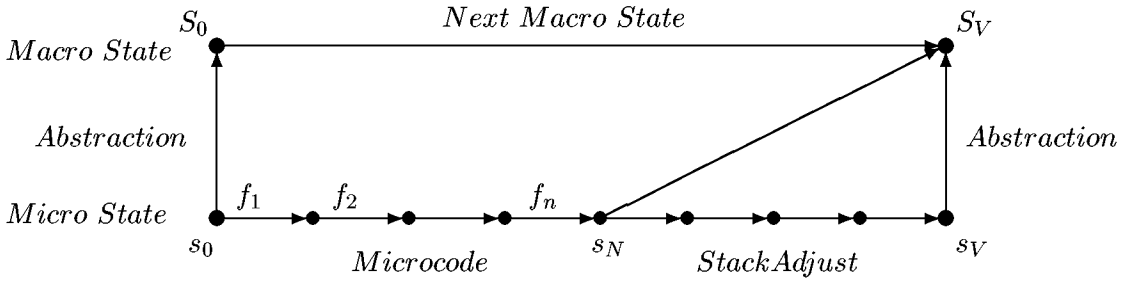


Figure 6.1: Overview of the Correctness Proof

The *micro correctness* proofs verify the correctness of the microcode at the microarchitecture (register-transfer) level, showing that the microcode executing on the microarchitecture specification satisfy several *micro correctness* lemmas describing how it changes the microarchitecture state. These proofs correspond to the lower sequence of micro instruction steps from microstate s_0 to s_N in Figure 6.1.

The *macro lift* proofs use these lemmas to show that the microcode correctly implements the

behavior specified at the macroarchitecture level. This is done by showing that the effect of mapping the initial microstate s_0 into macrostate S_0 via the *Abstraction* function and applying the `next_macro_state` function results in the same macrostate as mapping microstate s_N into macrostate S_V via the *Abstraction* function.

The micro steps from state s_N to s_V consist of the *stack adjustment* logic performed prior to each instruction to ensure that the correct number of operands are in the stack cache and that there is room in the stack cache for instruction results. Stack adjustments are not visible at the instruction set level and thus have no effect on the macrostate. Proof of the correctness of the stack adjust logic in support of the commutativity theorem only needs to be done once and involves showing that both the microstates s_N and s_V map to the same macrostate S_V via the *Abstraction* function.

6.1.2 Visibility Theorems

As described in Section 2.6, the visibility theorems define another set of properties to be proven about each instruction. The initial microstate s_0 in Figure 6.1 is assumed to satisfy these conditions. To achieve this, the visibility conditions must be shown to hold at processor initialization and at the end of each instruction at time t_V.

There are actually four visibility conditions to be shown for each instruction. The first requires that the instruction entry conditions be met, i.e., that the stack cache contains sufficient operands for the current instruction and that there is room in the stack cache for the instruction results. The second mandates that the code word pointed to by the program counter is actually loaded into the instruction register IR. The third ensures that the first line of microcode for the current instruction is loaded into the microcode register MC. Finally, the last condition guarantees that an invariant between the empty stack cache signal and a pointer SV into the stack cache is met. The visibility conditions are discussed in more detail in the following sections.

6.1.3 Invariant Theorems

Only one kind of invariant property (Section 2.6) has been proved about the AAMP-FV microcode, that the interrupt register correctly accumulates and remembers interrupts during instruction execution (see Section 6.2.1.4). While there are other interesting invariant properties that could be shown (e.g., that the overflow line is not asserted during instruction execution), the focus of this effort has been on the commutativity and visibility theorems. Proofs of additional invariant theorems may be undertaken in future efforts.

6.2 The Micro Correctness Proofs

This section discusses the *micro correctness* proofs, corresponding to the sequence of micro instruction steps from microstate s_0 to s_N in Figure 6.1. For each instruction, the result of executing its microcode is expressed as one or more lemmas, which are later used in the verification of the commutativity theorem. The proofs of these lemmas deal with data-dependent instruction execution times, branches, and loops, effectively separating these details from the macro correctness proofs and helping to compartmentalize the proofs, making them easier to manage. For engineers familiar with the AAMP-FV, these correctness conditions are quite easy to understand and completion

of the micro correctness proofs provide a high degree of assurance that the microcode is indeed correct.

Most of the AAMP-FV instructions have microcode that that is similar and can be verified by following a standard pattern. However, a few instructions have more complex microcode and require more sophisticated proof strategies. For this reason, the discussion of the micro correctness proofs is broken up into two parts, the proof of the standard AAMP-FV instructions and the proof of complex AAMP-FV instructions.

6.2.1 Standard AAMP-FV Instructions

The REFA instruction discussed in Section 4.2.4 is used to illustrate the micro correctness proofs for the standard AAMP-FV instructions. The micro correctness proofs for these instructions are organized into separate PVS theories, with the `REFA_micro_correct` theory being typical. Using a standard format for all instructions eased the creation of the micro correctness statement for each new instruction and simplified the macro correctness proofs.

6.2.1.1 The Micro Correctness Theory

Each micro correctness theory begins with the parameterization of the theory with the time `t0` and the assumptions the theory makes about that time, as shown below for REFA.

```
REFA_micro_correct[(IMPORTING time) t0: time]: THEORY

BEGIN

  ASSUMING

    IMPORTING correctness_predicates

       assume_current_op: ASSUMPTION current_op(t0) = REFA

       assume_visible: ASSUMPTION visible(t0)

       assume_normal_operation: ASSUMPTION normal_operation(t0)

  ENDASSUMING
```

Three distinct points in time are defined at the microarchitecture level for each instruction: `t0`, `tN`, and `tV`, corresponding to the microstates s_0, s_N, and s_V in Figure 6.1. The first of these, `t0`, represents the time at which instruction execution begins and each micro correctness theory is parameterized by this value. Time `t0` is treated as a constant within the micro correctness theory, but outside of this theory it can represent any time that satisfies the constraining assumptions. The assumptions are used in proving the micro correctness lemmas, and PVS generates proof obligations to show that each assumption holds whenever the micro correctness theory is instantiated.

The first assumption shown above asserts that the opcode at time `t0` is REFA. The proof obligation generated by this assumption is easily discharged at the next higher level in the proof structure.

The second assumption, `assume_visible`, brings in the visibility conditions discussed in Section 6.1.2. Its definition is

```
visible(t): bool =
  entry_conditions_met(t) and
    code_word_fetched(t) and
      instruction_loaded(t) and
        stack_invariant(t)
```

The first predicate, `entry_conditions_met`, holds if the stack cache contains an appropriate number of entries for the current instruction to execute. This is indicated by the microarchitecture when the stack adjust line is not asserted, i.e., the micro architecture component `ADJUST_F` is true. It is defined as

```
entry_conditions_met(t): bool = ADJUST_F(t)
```

The `code_word_fetched` condition requires that the instruction register contains the word from code memory pointed to by the program counter and is defined as

```
code_word_fetched(t): bool =
  IR(t) = read(code, UM(t), wordPC(t), MEMORY(t))
```

The `instruction_loaded` condition ensures that the microcode register (MC) contains the microcode ROM entry pointed to by the entry point of the currently requested operation.

```
instruction_loaded(t): bool =
  MC(t) = uROM(EP_OF(current_op(t)))
```

The last predicate in the `visible` assumption, `stack_invariant`, specifies an invariant between the signal `SKMT(t)`, which is true when the stack cache is empty, and the register `SV`, a pointer into the stack cache. Since the value of SV is maintained by the microcode, the micro correctness theory assumes that the previous instruction maintained this invariant.

The last assumption made by the micro correctness theory, `normal_operation`, ensures that the reset (`RST`) and end of built-in self test (`ENDB`) lines are not asserted during instruction execution.

```
normal_operation(t): bool =
  stays_low(ENDB)(t,end_of_current_instruction(t)) AND
    stays_low(RST)(t,end_of_current_instruction(t))
```

The `normal_operation` assumption thus asserts that the ENDB (end of builtin self-test) and RST (reset) signals stay low from time `t0` to `end_of_current_instruction(t0)`, a function that defines the time `tV` corresponding to micro state s_V in Figure 2.1. Note that while the micro correctness theory must make assumptions about the RST and ENDB lines, similar assumptions are not needed to deal with interrupts since the AAMP-FV only processes interrupts at instruction boundaries.

The next section of a micro correctness theory defines various points in time during the instruction execution. The uninterpreted function `end_of_current_microcode` defines the time at which the last line of microcode for the current instruction executes. Even though this function may be data dependent, it is always computable based upon the state of the processor at time `t0`. In the

REFA example above it is defined to have a value of `t0 + 1` at time `t0`. For notational convenience, `end_of_current_microcode(t0)` is aliased to `TC`.

```
REFA_TC: AXIOM
  end_of_current_microcode(t0) = t0 + 1

TC: time = end_of_current_microcode(t0)

tC: VAR {t:time | t = TC}

TN: time = end_of_current_microcode(t0) + 1

tN: VAR {t:time | t = TN}

tR: VAR {t:time | t >= t0 and t <= end_of_current_microcode(t0)}
```

Time `TN`, corresponding to the microstate s_N in Figure 2.1, is the time at which the instruction specific microcode completes execution and the results are loaded into the state registers of the AAMP-FV. For the REFA instruction, `TN` is equal to `TC` + 1. The variable `tN` is defined and used throughout the specification because it is helpful to use the correctness statements generated within this theory at a higher level as auto-rewrites. Defining `tN` to be a logical variable with only one possible value (`tN`) facilitates this during the proofs. The variable `tR` is defined to make it more convenient to define properties which hold for the duration of the instruction.

The rest of the micro correctness theory states the correctness lemmas to be proved. For example, the `REFA_SV_correct` lemma states that the stack occupancy when the REFA instruction is complete will be one less than the occupancy at the beginning of instruction execution.

```
REFA_SV_correct: LEMMA
  occupancy(tN) = occupancy(t0) - 1
```

The `REFA_STACK_correct` lemma states that the top element of the stack following the execution of the REFA instruction will contain the word read from the memory address address formed from the top two elements of the stack at the start of the instruction. The function `stack(t)(n)` returns the nth element of the stack cache at time `t`.

```
REFA_STACK_correct: LEMMA
  ((stack(tN)(0) = XD )
      WHERE
         WA  = (stack(t0)(1) o stack(t0)(0))^(23,0),
         XD  = read(t0)(WA))
```

The `REFA_UNUSED_correct` lemma defines what happens to stack cache elements not affected by the execution of the REFA microcode. Note that the index `j` ranges from 2 to one less than the occupancy of the stack cache at time `t0`. This is because the REFA instruction pops two words off the stack to create an address, then reads the word at that address and pushes it on the stack. This lemma states that any elements of the stack cache at time `t0`, other than the top two words used to construct the address, will be in the stack cache at time `tN` directly beneath the word read from memory.

```
j: VAR {k: nat | k < occupancy(t0) and k > 1}

REFA_UNUSED_correct: LEMMA
  stack(tN)(j-1) = stack(t0)(j)
```

Note that if the range of j exceeds its proper bounds, the error is caught during the proof of lemma REFA_UNUSED_correct. However, if the range of j fails to completely cover the full range, i.e., if it were specified as k > 2 the lemma REFA_UNUSED_correct would still be correct. In such cases, the error would not be caught until the macro lift proofs mapping the micro correctness lemmas into the other overall proof structure were completed. In fact, setting the range of j incorrectly was one of the more likely errors when creating the micro correctness lemmas. On at least two occasions, the macro correctness proofs revealed errors in setting the range of j that had not been discovered during the micro correctness proofs. This illustrates that while completion of the micro correctness proofs provides a high level of confidence in the microcode, completion of the macro lift proofs serves to check on the sufficiency of the micro correctness lemmas.

The remaining micro correctness lemmas are stated as a conjunction of conditions that must hold at completion of the instruction. For the REFA instruction, these consist of showing that the program counter is incremented by one, that the TOS, PAGE, LENV, and MASK registers are unchanged, that the processor remains in user mode, that memory is unchanged, and that the interrupt register is not cleared during the course of the instruction.

```
REFA_conjunction: LEMMA

  (PC(tN) = PC(t0) + 1)     &
  (TOSREG(tN) = TOSREG(t0)) &
  (PAGE(tN) = PAGE(t0))     &
  (LENV(tN) = LENV(t0))     &
  (MASK(tN) = MASK(t0))     &
  (UM(tN) = UM(t0))         &
  (MEMORY(tN) = MEMORY(t0)) &
   CLRI_accumulate(t0, TC)
```

These are specified as a single conjunction to increase proof efficiency. The proof of each micro correctness lemma requires that the PVS prover perform a symbolic simulation of the microcode using the microarchitecture model. By combining these into a single conjunction, PVS is able to utilize its caching facility to perform the simulation once, dispatching multiple correctness statements in parallel. This has a significant impact on proof speed.

The conjunctive style shown above was adopted mid-way through the AAMP-FV project. However, this change required that the form of the proofs that depend on this theory had to be changed as well. Rather than simply auto-rewriting the entire micro correctness theory, the conjunction had to be brought into the sequent, flattened, and each of the resulting propositions converted into a rewrite rule. This is illustrative of how the development of efficient proofs can cause unforeseen, and unintuitive, changes in the specifications.

6.2.1.2 The Micro Correctness Proofs

The actual proofs of the micro correctness lemmas rely heavily on the automatic rewrite capabilities of PVS. They typically involve performing auto-rewrites of all of the primary registers, wires, and signal definitions (DFFs, DFFRs, and CONNECTs) in the design, auto-rewriting any applicable bitvector rules, and performing an assert. This basic strategy is sufficient to dispatch most of the micro correctness proofs for the standard instructions.

Of course, each instruction is different and requires some some adjustment of the basic strategy. It becomes obvious through experience which registers and wires need to be rewritten for different

classes of instructions. For example, instructions which do not access memory (such as the ADD instruction) do not need to have the theories associated with the main memory installed. There is a trade-off, however, between minimizing the number of automatic rewrites and maximizing the reusability of the proof script.

More difficult to anticipate are the case splits required in the proofs of different instruction. Often, it is not obvious what specific values a branch in the proof should be based upon until one has reviewed a failed proof, so many of the variations on the basic proof script were developed through trial and error. It would be difficult to develop a single proof strategy that would work well on all of the standard instructions.

6.2.1.3 Proofs of Visibility Properties

In order to ensure the proper setup for the next instruction, there are several properties, in addition to the micro correctness lemmas, that must hold at the end of the current instruction. Most of these relate to showing that the visibility assumption for the next instruction can be met, e.g., ensuring that stack occupancy requirements are met and that the microcode is correctly loaded. These functions are handled by the stack adjustment logic common to all instructions. The theory `NEXT_micro_correct` is imported by the micro correctness theory for the current instruction. Dispatching the type correctness conditions (TCC) generated by importing this theory guarantees that the microcode for the current instructions satisfies the assumptions needed by the stack adjustment logic. In this way, the verification of the microcode for the current instruction and the verification of the stack adjustment logic can be combined to complete the bottom line of the commuting diagram in Figure 6.1 corresponding to states s_0 to s_V.

6.2.1.4 Proofs of Invariant Properties

Some properties of the AAMP-FV are best proven entirely at the microarchitecture level, without trying to relate them to the macroarchitecture specification. For example, the AAMP-FV captures interrupts raised while an instruction executes and holds them in the interrupt register pending the end of the current instruction. Precisely defining this behavior is difficult in the macro architecture since it has no real notion of time, just the state of the processor and memory before and after the instruction. To ensure that interrupts are correctly handled, the micro correctness theory imports another theory, `INT_REG_micro_correct`, that states that the interrupt register correctly accumulates and remembers new interrupts. At the macro architecture level, this is weakened to state that under normal operation (i.e., ENDB and RST remain low) the interrupt register remains unchanged if no interrupts are raised during the instruction. This invariant has been proven to hold during the instruction specific microcode of each instruction. [1] The proof that it holds for the stack adjust logic has not been completed.

6.2.2 The Complex AAMP-FV Instructions

Some of the AAMP-FV instructions, such as the CALL and IMPY instructions, are sufficiently complex that the strategy described above was insufficient for the verification of their microcode. To deal with these instructions, the specification of the micro correctness lemmas was split into several theories and more complex strategies were used in their proof.

[1] With the exception of the CLRI instruction, which clears the interrupt register.

The first strategy consisted of dividing the microcode up into sections that perform some well defined function, then "gluing" the verification of these sections together, much as was done in the REFA instruction when combining the verification of the REFA specific microcode to that of the stack adjust logic. The second strategy consisted of proving key properties at the lowest level, then using those results to discharge more general proof obligations at the next level up. A third strategy, used in the CALL instruction, allowed the prover to step forward through the symbolic execution of the microcode one cycle at a time.

6.2.2.1 The CALL Instruction

The AAMP-FV CALL instruction transfers execution control to a called procedure and provides for the passage of parameters, allocation of dynamic storage on the stack for the procedure's local variables, and the creation of the stack mark. As such, it is one of the more complex AAMP-FV instructions.

Its microcode is divided into three parts identified as the setup, adjust, and main segments. The setup segment performs the initialization of several registers in preparation for execution of the rest of the instruction. The adjust segment ensures that the stack cache is empty (for efficiency, this is done by a special segment of microcode rather than by the normal stack adjust logic). The main segment then performs the real work of the CALL instruction by writing the stack mark, saving the current processor state, and performing a context switch.

Organizing the correctness theories for the CALL instruction around this structure makes it possible to treat the microcode as three simple segments with minimal branching. This simplifies the symbolic simulation of the microcode by reducing the number and complexity of the case splits that need to be introduced during the proof.

It was also useful to separate the verification of each segment of microcode from the representation of the results used at the next higher level. At the lower levels, the proofs could be made more efficient by stating the properties to proven in the conjunctive form discussed in Section 6.2.1.1, thus exploiting the caching capabilities of PVS to symbolically execute the microcode once for all properties rather than once for each property to be proven. At the next higher level, these same properties were restated as individual lemmas suitable for use as automatic rewrite rules. These lemmas were easily discharged using the conjunctive form proven at the next lower level.

These two strategies, breaking the microcode up into three segments and reformulating the correctness properties to support the next higher layer, resulted in the proof structure shown in Figure 6.2. The `predicate` theories are where the difficult part of the proofs, i.e., the symbolic execution exploiting the caching capabilities of PVS, are located. The `correct` theories are cosmetic restatements of the `predicate` theories expressing the properties as lemmas better suited for rewriting.

For example, the `CALL_setup_predicate`, `CALL_adjust_predicate`, and `CALL_main_predicate` theories are where most of the work in proving the CALL micro correctness lemmas occur. Each segment of microcode is symbolically executed once in these three theories and used to prove the conjunction of the micro correctness lemmas for each segment. Their parent theories, `CALL_setup_correct`, `CALL_adjust_correct`, and `CALL_main_correct`, break these statements into several lemmas suited for use as automated rewrite rules. The `CALL_micro_predicate` theory combines these three theories together to create the "end-to-end" version of the micro correctness lemmas for

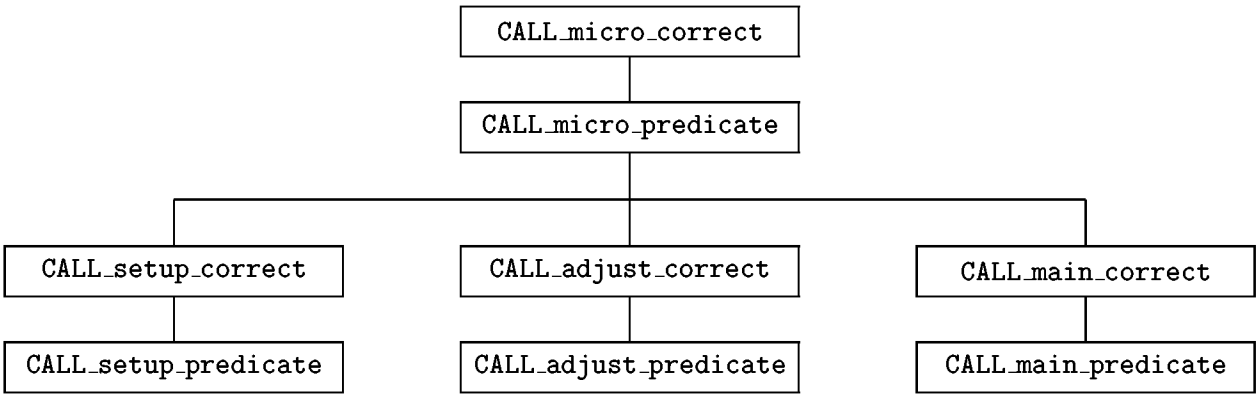


Figure 6.2: CALL Proof Structure

the CALL instruction. Finally, the `CALL_micro_correct` theory restates these results in the form expected by the macro lift proof of the CALL instruction.

Another technique used in the verification of the CALL microcode was to force the PVS prover to evaluate the microarchitecture state by stepping forward one cycle at a time. This had two main advantages. First, it improved proof efficiency by allowing the proof to delay case splits, thereby making better use of PVS's caching facilities. Second, it made it easier for a human to monitor the progress of the proof.

Normally, the PVS prover attempts to prove a microarchitecture property postulated at time `T` by recursively rewriting it as a property postulated at time `T - 1` until the start of the instruction at time `T0` is reached. PVS then works its way forward in time, reducing expressions whenever possible. This is a natural consequence of defining the microarchitecture state at time `t + 1` as a function of its state at time `t`.

While this technique worked well enough for the standard instructions, the microcode for the CALL instruction is complex enough that the expressions generated while recursing to time `T0` became difficult to manage. Worse, any errors in the specification amplified the problem, usually without providing any clues of what the error was.

To address this, a method was developed to force the PVS prover to start with the microprocessor state at time `t0` and step forward, computing the new microprocessor state at each micro cycle until the desired time was reached. This strategy also provided a simple mechanism to let the verification engineer examine the micro state at each time. This made it much simpler to spot problems caused by errors in the specification.

To perform micro stepping, the `step` function was defined as shown in Figure 6.3. The `step` function takes two arguments, a starting time `t` and a completion time `TN` and returns yet another function, `step(t,TN)`. This function takes two predicates over the micro state, `A`, defining a predicate that is to hold throughout instruction, and `P`, defining a predicate that is to hold at the end of the instruction, and returns a boolean value.

When using the `step` function in a proof, `step` is expanded at time `t` to expose its definition. Providing `t` is less than `TN`, The PVS prover evaluates `A` at time `t`. Assuming `A` is some function

```
step: THEORY

BEGIN

  IMPORTING micro_state

  B          : VAR bool
  A,P        : VAR [time -> bool]
  t,t0,tR,TN : VAR time
  S          : VAR micro_state

  split(S,B) : bool = B

  step(t,TN)(A,P): RECURSIVE bool =
    IF    (t >= TN)
    THEN  P(TN)
    ELSE  A(t) & split(uState(t),step(t+1,TN)(A,P))
    ENDIF
  MEASURE TN - t

  step_accumulate: LEMMA (t0 <= tR) & (tR < TN) & step(t0,TN)(A,P) => A(tR)

  step_step   : LEMMA (t<TN) => step(t,TN)(A,P) = step(t+1,TN)(A,P)

  step_induct : LEMMA step(TN - t,TN)(A,P) => P(TN)

  step_rewrite: LEMMA step(t0,TN)(A,P) => P(TN)

END step
```

Figure 6.3: Step Function

of the micro state at time `t`, this forces the PVS prover to evaluate the micro state at time `t`, and in the process, place the micro state at time `t` in its cache. If `A(t)` holds, the prover `step` proceeds to evaluate `split`. The `split` function is actually dummy function. Its sole purpose is to allow the proof engineer to examine a representation of the micro state at time `t`, `uState(t)`. If the micro state appears reasonable, the proof continues by expanding `step` at time `t + 1`. This process continues until `t` is equal to or greater than `TN`, at which time the correctness state `P` is evaluated at time `TN`. The `step_accumulate` and `step_rewrite` lemmas are useful in establishing the desired final results.

As mentioned earlier, this approach has two advantages. First, the verification engineer can monitor the progress of the proof by examining `uState` at each step. The function `uState` can be defined to be whatever representation of the microarchitecture state is the most helpful. This makes it much simpler to detect when a proof has gone astray due to a specification error. Second, the PVS prover caches the intermediate results as the proof steps forward. By delaying case splits in the proof as long as possible, PVS can use the cached results when evaluating new instances of the micro state.

6.3 Proof of the Stack Adjustment Logic

From time t_N to t_V, a common piece of microcode, the *stack adjustment* routine is executing. This routine ensures that the requisite number of arguments are read into the stack cache in preparation for the next instruction or that sufficient words are written out to memory to make room for the computed result. Since this microcode is common to all instructions, its correctness only needed to be proven once. Combining the proof of the micro correctness conditions with the proof of the stack adjustment routine completes the lower arm of the commuting diagram from microstate s_0 to s_V in Figure 2.1. The stack adjustment logic was verified by SRI.

6.4 The Macro Lift Proofs

Once the detailed behavior of the microcode is verified at the microarchitecture level, the overall proof of correctness is completed by the macro lift proof that "lifts" the micro correctness conditions to the macroarchitecture level. In Figure 6.1 this corresponds to the arms mapping microstates s_0 and s_N into macrostates S_0 and S_V via the *Abstraction* function and the `next_macro_state` function that transforms macrostate S_0 into macrostate S_V.

6.4.1 The Abstraction Function

The abstraction function of Figure 6.1 is defined in theory `ABS` shown in Figure 6.4. This function constructs the macroarchitecture state seen by the application programmer from the microarchitecture state. For the most part, this simply consists of equating, or "lifting", components directly from the microarchitecture state to the macroarchitecture level. For example, `ABSpc` simply returns the value of the microarchitecture program counter register at time `t`.

One exception is the top of stack, or TOS, register seen by the application programmer. The microarchitecture `TOS` register actually points to the topmost (i.e., smallest address) word of the process stack *maintained in memory*, while the application programmer sees the "logical" TOS created by including the number of words maintained in the stack cache to this address. As a result, the logical TOS, `ABStos`, is defined to the microarchitecture `TOS` minus the current occupancy of the stack cache (recall that the process stack grows towards decreasing memory addresses).

A more complex exception is the application programmer's view of memory, which is a blending of actual memory and the stack cache. The application programmer's view, `ABSmem`, returns the original memory function defined in Figure 4.4, except that it substitutes the contents of the stack cache for words in memory locations overlaid by the stack cache. To obtain the correct index into the stack cache, `ABSmem` converts an address `addr` into an offset from the top of the stack cache and subtracts that offset from the occupancy of the stack cache.

Finally, the function `ABS` uses the individually defined components of the abstraction function to create the PVS record representing the macrostate.

6.4.2 The Macro Correctness Statement

For each instruction, a lemma is created in PVS defining what it means for that instruction to be "correct". The statement of correctness for the REFA instruction is given in lemma `REFA_macro_correct` in Figure 6.5. Here, `t0` is the time at which the instruction begins and `TN` the time at

```
ABS: THEORY
  BEGIN

    IMPORTING AAMP_FV_microarchitecture
    IMPORTING macro_state

    t: VAR time

    %-----------------------------------------------------------------------
    % Lift the internal registers and flags to the macro level.
    %-----------------------------------------------------------------------
     ABSpagereg(t) : register   = PAGE(t)
     ABStos(t)     : register   = TOSREG(t) - occupancy(t)
     ABSpc(t)      : address    = PC(t)
     ABSlenv(t)    : register   = LENV(t)
     ABSum(t)      : bool       = UM(t)
     ABSmask(t)    : byte       = MASK(t)
     ABSintreg(t)  : byte       = INT_REG(t)

    %-----------------------------------------------------------------------
    % Lift external memory and the stack cache to the macro level.
    %-----------------------------------------------------------------------
     ABSmem(t)(cd, ue: bool)(addr: address): word =
       IF in_stack_cache_area(t,cd,ue,addr) THEN
         stack(t)(occupancy(t) - offset(t,addr))
       ELSE
         MEMORY(t)(cd,ue)(addr)
       ENDIF

    %-----------------------------------------------------------------------
    % ABS returns the macrostate as a function of the microstate
    % at time t, "lifting" the visible microstate to the macro level.
    %-----------------------------------------------------------------------
     ABS(t): macro_state = (# mem     := ABSmem(t),
                              tos     := ABStos(t),
                              pc      := ABSpc(t),
                              lenv    := ABSlenv(t),
                              um      := ABSum(t),
                              mask    := ABSmask(t),
                              intreg  := ABSintreg(t) #)
   ...                      ...                    ...
END ABS
```

Figure 6.4: Abstraction Function

which the instruction ends. `ABS(t0)` and `ABS(TN)` correspond to the macrostates at the start and completion of the REFA instruction, respectively. This lemma states that if the opcode at time `t0` is REFA and the macro restrictions apply, then the macrostate created by applying the abstraction function `ABS` to the microstate at time `TN` is the same as that obtained by applying the `next_macro_state` function to the macrostate created by applying the abstraction function to the micro state

```
REFA_macro_restrictions : AXIOM
  current_opcode(ABS(t0)) = REFA =>
    macro_restrictions_apply(ABS(t0)) =
      (not_stack_cache_address(ABS(t0))(WA) &
       pc_not_in_cache_region(t0))
         WHERE WA = (top(ABS(t0),1) o top(ABS(t0),0))^(23,0)

REFA_macro_correct : LEMMA
  current_opcode(ABS(t0)) = REFA &
     macro_restrictions_apply(ABS(t0)) =>
        next_macro_state(ABS(t0)) = ABS(TN)
```

Figure 6.5: PVS Correctness Statement for the REFA Instruction

at time `t0`. The `next_macro_state` function was discussed in Section 4.2.4 and is given for the REFA instruction in Figure 4.6 on page 23.

The `macro_restrictions_apply` predicate defines the conditions that must be met for the instruction to be well defined under the abstraction we have defined. For most instructions, this simply consists of the restriction `pc_not_in_cache_region`, which is `true` except when the data and code regions overlap in memory. If the data and code regions overlap, then the restriction also requires that the memory location addressed by the program counter not lie within the stack cache region. Instructions that directly reference memory, such as the REF and ASN instructions, need additional restrictions. For example, they require that the memory word(s) being referenced cannot lie in the vicinity of the stack cache. The `REFA_macro_restrictions` axiom assigns the `macro_restrictions_apply` predicate this interpretation when the current instruction is REFA. For instructions such as REF24, that calculate the address from bytes drawn form the instruction stream, it is further necessary to require that the instruction bytes do not overlap with the scache region if the code and data regions overlap in memory.

The `REFA_macro_restrictions` axiom repeats the constraints included in the definition of the REFA instruction given in Figure 4.6. When the macroarchitecture was specified, it wasn't apparent that a separate specification of `macro_restrictions_apply` would be needed during the proofs. In future efforts, it would make more sense to specify `macro_restrictions_apply` once for each instruction and incorporate this definition into the macro architecture specification.

6.4.3 The Macro Lift Proofs

The actual macro lift proofs all follow the same overall pattern, yet differ enough that each requires individual attention. The proof for the REFA instruction is typical. It consists of 66 PVS prover commands.

The first part of the proof consists of 36 prover commands. These instruct PVS to generate automatic rewrite rules from several PVS theories such as the bit vectors, pull in the macroarchitecture definition of the REFA instruction, pull in the microarchitecture correctness lemmas for the REFA instruction, pull in the abstraction function, and then expand the correctness statement shown in Figure 6.5 using all of the above as rewrite rules. At this point, PVS is able to conclude

through auto-rewrites that the correctness statement holds for all components of the macrostate except for memory.

As discussed in Section 4.4, memory consists of up to four memory spaces indexed by the code/data and user/exec lines. The proof first splits on whether the memory space is for code or data. Since the REFA instruction does not change code memory, PVS is able to show that the correctness condition holds for this branch with a single ASSERT prover command. Next, the proof splits on whether the remaining data memory space being considered contains the process stack or not. Since the data memory space that does not contain the process stack is not changed by the REFA instruction, PVS is able to show that the correctness condition holds via application of several ASSERT and GROUND commands.

The proof that the correctness condition holds for the memory space containing the process stack is the most complex portion of the proof and consists of 22 prover commands. These split the proof into three branches, first considering the portion of the memory space lying above (i.e., at a lower addresses) than the new logical top (TOS) of the process stack, the specific word pointed to by the new TOS value (which contains the word read from memory and placed on the top of the stack), and the unchanged portion of memory lying below (i.e., at higher addresses) than the new logical top of stack.

Execution of the REFA macro correctness proof requires approximately 618 seconds (run time) on a SPARC 20 workstation.

Chapter 7

Lessons Learned

Many insights have been gained during the AAMP5 and AAMP-FV projects. This chapter discusses some of the more important lessons.

7.1 Technology Transfer

Learning how to specify and verify formally the AAMP5 and AAMP-FV has been a long and challenging process. The paradigm followed on both projects has been for SRI to develop the initial approach on a few examples, then have Collins apply it to several examples, refining and generalizing the approach. Throughout the project, things that appeared overwhelming at the start were eventually mastered and reduced to routine, repeatable steps. These gains were achieved by 1) direct reuse of earlier specifications, 2) creating examples of how best to specify in PVS features of the AAMP family, and 3) consolidating in the same individuals an understanding of both the AAMP family and the expertise of how to use PVS.

An example of the direct reuse of earlier specifications is the bit vectors library developed during the AAMP5 project. This library was used without significant modification on the AAMP-FV project and greatly reduced the time needed to create the *specifications* of the micro and macro architectures. While doing the *proofs* it became clear that the library needed to be supplemented with a large number of lemmas that could be invoked as rewrite rules (Section 7.3). This points out that even established libraries will need to be continuously enhanced.

A good example of learning of how best to specify in PVS features of the AAMP family is provided by the specification of the AAMP-FV instruction set. In specifying the AAMP5 instruction set, a *constructive* style of specification, in which each instruction was specified by stating a function that directly transformed the macro state, was originally chosen. Later, it was recognized that a more *descriptive* style, in which the effect of each instruction was described by giving the macro state before and after the instruction, more in the form of pre and post conditions, was more useful, easier to write, and simpler to review [27, 37, 36]. Changing to the more descriptive style played a major role in reducing the cost of specifying the AAMP-FV instruction set. As a result, specifying each instruction at the macroarchitecture level is now quite routine, usually requiring less than an hour.

The most dramatic gains in efficiency were achieved by consolidating in the same individuals an understanding of both the AAMP family and PVS. For example, creating the AAMP5 mi-

croarchitecture specification took over 1,000 man hours to complete, while creating the AAMP-FV microarchitecture specification took less than 120 hours. This occurred because many of the details of how to specify the architecture were well understood from the AAMP5 project and the AAMP-FV microarchitecture expert was able to write similar specifications for the AAMP-FV.

Much less experience with doing proofs of correctness had been transferred to Collins during the AAMP5 project. As a result, a significant portion of the AAMP-FV project was involved with mastering this technology and refining it so that it could be repeated consistently. While there are still improvements that can be made, for many instructions, this process has also become routine. Fifty-four of the AAMP-FV's 80 instructions have been formally verified, at an average cost of about 38 hours per instruction. These proofs are similar in that they based on a symbolic execution of the the microcode. On future efforts, we believe this cost could be cut in half because the proof method is now better understood.

Many of the remaining 26 AAMP-FV instructions consist of instructions such as multiply, divide, shift, call, and return that are implemented with considerably more complex microcode. Simple symbolic execution of the microcode fails here because the expressions generated during the proof become too large. As a result, more efficient approaches had to be developed. For example, proof of the multiply and divide instructions are based on identification of a loop invariant that holds as each microinstruction is executed and that can be used in a proof by induction. At this time, examples of how to perform these proofs have been developed by SRI, but these methods have not yet been widely applied and refined by Collins.

In summary, the role of SRI has been to develop the first, depth-first examples, while Collins performs the breadth-first application and generalization of those examples. This experience is transferred as reusable libraries, examples tailored to the AAMP family, and consolidation of both AAMP domain knowledge and PVS expertise in the same individuals. Costs during development of these methods are quite high, but drop significantly, even by an order of magnitude, as the technology is mastered. At the current time, specification of the macro and micro architectures is almost completely performed by Collins, proof of the more routine instructions is well understood, and techniques to prove the more complex instructions are being transferred to Collins.

7.2 Development of Domain Specific Libraries

Although one of the contributions of the AAMP5 project was the development of an extensive library specifying the properties of bit vectors (i.e., sequences of bits such as bytes and words), manipulation of bit vectors proved to be a constant challenge while doing the AAMP-FV proofs. In retrospect, this shouldn't have been surprising. Most of the AAMP5 project, particularly for Collins, focused on specification of the micro and macroarchitectures. Consequently, while the the bit vector library developed on the AAMP5 project worked admirably for *specification* of the AAMP-FV, more work had to be done to use it for *verification* of the AAMP-FV microcode. Most of this consisted of the development of lemmas about the bit vectors that can that can be used as automatic rewrite rules. Also, SRI has incorporated many of the key properties of bit vectors in decision procedures in PVS.

While considerable progress has been made in extending the bit vector library, more work still needs to be done. It is not yet clear that the best model for the bit vectors has been chosen, that the best set of operations for constructing and destructing bit vectors have been defined, and that the best set of rewrite rules have been developed. Precisely how decision procedures for bit vectors

should be incorporated into PVS and how they should be supplemented with rewrite rules also needs more attention.

7.3 Proof Robustness

One of the main problems encountered during the AAMP-FV project was that proofs completed during the earlier part of the project would break as the specifications were changed to complete proofs encountered later in the project. Usually, this occurred as proofs uncovered defects in the specification, additional proof obligations are identified, or required changes in the specification to facilitate the proof. For example, midway through the AAMP-FV project, we realized that the memory model, which was parameterized with whether the code/data and user/exec lines were used to partition memory, could be generalized to the unparameterized version discussed in Section 4.1.1. This allowed us to verify the AAMP-FV microcode regardless of how memory was configured, reducing the number of proofs by a factor of four. However, it also required considerable effort to go back and generalize the proofs already completed using the old memory model.

In other cases, proofs would be encountered that could be simplified by making small changes to the specification, but these changes would break existing proofs. Less frequently, actual errors in the PVS specifications would be found in later proofs. On a few occasions, upgrades to PVS broke existing proofs. All of this points to the need to automate the proof process as much as possible on large projects. This is particularly true of industrial projects, where constant change is the rule rather than the exception. Two approaches suggest themselves, designing proofs to be as robust as possible and increasing the amount of automation PVS can bring to bear on a problem.

Designing proofs to be a robust is a topic in itself worthy of further work. However, a few guidelines have surfaced during the AAMP-FV project. Automation is usually desirable because it makes a proof less sensitive to minor changes in the specifications. For example, one of the most useful techniques used on the AAMP5 and and AAMP-FV projects was to develop rewrite rules that exploit PVS's rewriting capabilities to automate as much of the proof as possible.

There are also a variety of heuristics that can be used to make proofs less fragile. Generally, the less application specific information provided in a proof, the less fragile it will be. For example, a `ground` command is generally more robust than performing a `case` split (which requires identification of the specific predicate to split on) followed by `assert` commands. However, the latter is often more efficient and may even be necessary to complete a long proof.

Using care in how application specific information is used can also make a proof more robust. For example, when phrasing a case split, the most obvious choice, based on just looking at the sequent, may be more fragile than another form more closely tied to the actual problem. For example, in the macro lift proofs, case splits could often be stated in terms of either the microarchitecture or the macroarchitecture, but stating them from the macroarchitecture view made them less sensitive to changes to the microarchitecture specification. This was particularly true if automated rewrite rules immediately rewrote the case split predicates in terms of the microarchitecture. Since the macroarchitecture specification was less likely to change than the microarchitecture specification, this resulted in more robust proofs.

Another guideline is to avoid the use of specific formula numbers in a proof (e.g., `assert -3`), as these identifiers often change. Forms such as `assert -` or `assert *` are more robust.

The use of parameterized theories often made the proofs more difficult than necessary, largely because of the need to instantiate the parameters with specific values during the proof process. On

several occasions, we replaced parameterized theories with unparameterized versions after dealing with the consequences while doing proofs.

The other approach to increasing the robustness of proofs would be to enhance PVS to bring more automation to bear on a problem. The `ground` command mentioned earlier is an example of a powerful PVS command that improves the robustness of a proof, though often at the cost of additional CPU time.

A facility to tag antecedent, consequent, and hidden formulae would also be helpful, allowing one to write prover commands of the form `assert` *tag*, where *tag* is a name earlier associated with the formula of interest. In particular, it would be helpful to tag formulas with names at the time there are hidden so that they can later be revealed using that tag. One of the most fragile parts of the macro lift proofs was a lemma that was introduced early in the proof, then hidden so that it would not be affected by automatic rewrite rules, then revealed later in the proof. Since revealing it required that its position among the hidden rules be given explicitly, the `reveal` statement was a frequent source of trouble.

One thing that will help greatly is the introduction of more speed. Some of the micro correctness proofs for the AAMP-FV take more than a day to complete on a Sparc 20 workstation, and correcting such proofs due to minor changes to the specifications is a frustrating task. An order of magnitude improvement in performance would go a long ways towards mitigating this. This is not as outrageous as it might sound: many of the proofs completed on the AAMP-FV probably could not have been done on the technology available at the start of the AAMP5 project. While more speed will not actually make proofs more robust, it will mitigate the consequences of fragility.

7.4 Exploiting Modularity

PVS supports modularity through constructs such as theories, parameterization of theories, importing (making a theory visible), and exporting (hiding parts of a theory). In addition, PVS also provides facilities for files, which hold one or more theories, and libraries, which are similar to file directories. Breaking the AAMP5 and AAMP-FV specifications down into many small, logically related theories was a great help in organizing and structuring these large specifications. If only to minimize the amount of typechecking that must be redone when modifying a specification, careful structuring of a specification is worth the effort invested.

Considerable work has been done by the software community to develop heuristics that maximize *cohesion*, or the degree to which the constructs in a module are related, and to minimize *coupling*, the degree to which modules depend on each other. Besides producing well organized specifications, this places together portions of the specification that are likely to change together and minimizes the extent to which changes affect other modules.

The AAMP5 and AAMP-FV projects were both unusual in that they had stable informal specifications from which the formal specifications were developed. Even so, change caused by errors found in the specification and to facilitate the proofs became one of the largest costs associated with the project (Section 7.3). In most industrial applications, constant change is the norm rather than the exception. This suggests that better techniques for managing and mitigating the effect of change need to be incorporated into tools such as PVS if they are to used in parallel with the development of industrial systems. As with engineering proofs for robustness, this can be achieved both by structuring theories to be robust in the face of change and by enhancing PVS.

A specification can be made more robust by using *interfaces* as firewalls. All the definitions that other theories can reference are placed on the interface, while supporting definitions are encapsulated or hidden behind the interface. By encapsulating definitions that are likely to change, the effect of change can be mitigated. Interfaces can be established around a single theory or a group of theories. In a system such as PVS, where one wants to reason formally about a specification, it is necessary to place key properties on the interface as well as type definitions. For example, one might want to place on the interface certain lemmas that can be invoked during the proof process, but hide behind the interface a constructive specification that exhibits these properties.

This sort of interface was placed around the bit vectors library during development of the AAMP5 and AAMP-FV project. As lemmas were developed that could be used as rewrite rules for the bit vectors, they were added to a "shell" of theories sitting directly above the bit vectors. Eventually, they became complete enough that the original constructive specifications were no longer needed for either the specifications or the proofs. However, the constructive specification was still used to prove the lemmas, thereby establishing their consistency. Later, a fundamental change to the constructive specification was proposed. Researchers at NASA Langley were able to implement this change in a few days and incorporating it into the AAMP-FV specification merely required proving that the lemmas on the interface still held.

This interface was enforced on the AAMP-FV project by convention. The actual mechanism for enforcing such interfaces in PVS is the `EXPORTING` clause, which exports from a theory names that are to be made available to the rest of the context. The `EXPORTING` clause was not used on either the AAMP5 or AAMP-FV project, in part because the PVS Reference Manual discourages its use. However, our experiences suggest that more care needs to be given to mitigating the effect of change in large verification efforts. Further research may suggest the addition of other mechanisms than just the `EXPORTING` clause.

There are also enhancements to PVS that would help to manage change. Currently, a change to any theory requires all theories that import that theory be retypechecked and their proofs rerun, whether they are affected by the change or not. A typechecker that only rechecked those portions of the specification affected by a change could greatly reduce the amount of time spent retypechecking and rerunning proofs. If would also be helpful to be able to determine the scope of the theories affected by a change before actually making the change.

It is likely that additional constructs to help minimize the impact of change will need to be added to PVS. For example, the *library* feature was recently added to PVS that can be used to group theories into libraries that are reasonably stable, but additional documentation is needed on how to use this feature.

A number of heuristics for determining the quality of software interfaces have been developed over the years. Embley and Woodfield have published a method for quantifiably assessing the quality of the cohesion exhibited by an Ada package [15]. Similar concepts could be developed for specifications written in PVS. Given such heuristics, it is not difficult to envision the addition of automated *design critics* to PVS that could automatically flag poor structuring choices.

7.5 Support for Product Families

Formal specification and verification is an expensive process, making it important to amortize its cost over several projects. One way to do this is to develop reusable libraries such as the bit vectors. Another is to *scavenge* portions of one specification that can be copied, modified, and used

in another application. Portions of the AAMP-FV specification were profitably scavenged from the AAMP5 project.

However, if it is known from the outset that a family of related products is going to be developed it makes sense to plan for the reuse of specifications and proofs in a more systematic manner, commonly referred to as *domain*, or *product family*, engineering. In this approach, a core set of common features are developed for use by all members of the product family. Members of the product family are then instantiated by supplementing this core specification with details specific to that member. As members are created, those features common to several variants are generalized and folded back into the main, or domain definition.

Domain engineering is closely related to modularization of specifications (Section 7.4) since one is concerned with identification of commonalities and differences between members of the product family. Portions of the specification known to be stable and common are incorporated into the core of the domain definition, while those likely to change are placed in variant definitions. In this respect, PVS already contains many of the *specification* constructs necessary to support a product family. However, it is not clear that all the features needed to support reuse of *proofs* exist. For example, in the AAMP-FV project all the macro lift proofs were variations of the same basic proof structure. Yet is was impractical to specify this structure once and instantiate it with the details needed for each distinct instruction.

Creating a domain definition is obviously more difficult than specifying a single member of the family and would not have been appropriate for exploratory projects such as the AAMP5 or the AAMP-FV. However, as the technology matures from exploratory to commercial application, techniques for the systematic, planned reuse of specifications and proofs will become essential.

7.6 Importance of the User Interface

Another lesson that was not fully appreciated prior to the AAMP-FV project was the importance of the user interface. During the AAMP5 project, SRI added to PVS the ability to graphically display the import chain for a family of theories and the tree structure of a proof. Displaying the import chain for the AAMP-FV specifications immediately pointed out unnecessary and undesirable imports that had gone completely unnoticed. In the same way that being handed a map changes one's perception of the forest, it brought out patterns that had not been seen when looking at the specifications a theory at a time.

The importance of being able to visualize the tree structure of a proof was dramatically demonstrated while completing the macro lift proofs. Once this facility was provided, it quickly became the norm to display the proof tree adjacent to the PVS prover window. Midway through the project, several very long proofs (taking over a day to execute) were being developed when a bug was uncovered in the software displaying the proof tree. After a week of effort, it became apparent that the proof was too difficult to understand without the graphic display of the proof tree. The bug was reported to SRI and several similar proofs that did not invoke the bug were completed. Later, when the correction from SRI was installed, the proof of the offending instruction was completed in a few hours. In short, the ability to visualize the overall structure of the proof and one's position in it played an essential role in managing the proof.

As larger proofs and specification are undertaken, the need for interfaces that assist the user in understanding their structure and navigating within them will continue to grow. Even now, tools that made it easier to visualize the overall AAMP-FV specification, navigate through it, zoom in on

portions of interest, and better anticipate the effect of changes would be very helpful. Undoubtedly, there are valuable improvements to the user interface that haven't even been envisioned.

7.7 What Needs to be Proven?

Most formal verification projects have been performed in the context of research or exploratory efforts. Exploiting this technology for commercial use will force it to move closer to an engineering discipline. However, an important aspect of an engineering discipline is that solutions must be cost-effective [32]. This tension between research and engineering manifested itself during the AAMP-FV project in the form of a simple question: What needs to be proven?

For microprocessor verification, one end of the spectrum requires a complete proof that the transistors correctly implement each instruction. While an exciting prospect, this approach was not chosen for the AAMP-FV simply because sufficient funding didn't exist. Previous experience building members of the AAMP family had shown that the microcode is the most likely source of error. For this reason, both the AAMP5 and the AAMP-FV projects focused on microcode verification.

Besides providing a very high level of assurance in the correctness of the microcode, this had many indirect benefits. Since verification of the microcode is based of symbolic execution of it on a specification of the microarchitecture, it was necessary to specify the microarchitecture formally. This process, along with proof of the microcode, provided a very detailed review of the microarchitecture that that was likely, if not guaranteed, to find errors in its design.

As the project evolved, it became clear that the hardware engineers gained the greatest assurance from completion of the micro correctness proofs, and were not very concerned with the macro lift proofs. Yet on several occasions, completion of the macro lift proofs pointed out pieces of the microcode that had not been verified. For example, a subtyping error twice caused the micro correctness proof to overlook the correctness of a word in the process stack. Nothing was actually wrong with the microcode, the proof just wasn't completed.

At the same time, completing the macro-lift proofs provided an important validation of the instruction set, or macro architecture, specification. More importantly, completion of the proofs forced us to model several subtleties that had been omitted from the original specification. To create this specification, both the micro correctness and macro lift proofs were essential.

But what if time or money doesn't exist to complete every proof? Is there value in only completing some of them? One of the authors is now using PVS routinely to execute microcode symbolically on his current project. This gives him much greater confidence in the correctness of the microcode, without incurring the cost of full formal verification early in the project. While these partial proofs may be extended to full proofs of correctness before the project ends, that decision will be based on an economic judgement of the costs versus the benefits.

Some lemmas seem obviously true but can still be difficult to prove. Can such proofs be skipped? The risk involved may depend on why the lemma was created in the first place. It was not uncommon when verifying a particular instruction to reduce the entire proof to the equivalence of two moderately complex bit vector expressions. Often, this could be stated as a lemma and proven independently. While some of these appeared obviously true, if there had been an error in the microcode, it would have been distilled down and buried in that very lemma. This situation actually arose on the AAMP-FV project when verifying the ABSD instruction, emphasizing that

the question of what needs to proven is not merely one of aesthetics, but a practical concern deserving careful thought as costs are balanced against benefits.

What if the microcode is verified through proofs once, but later a change made to the specification to facilitate another proof breaks the proof? Does the verification of the microcode break with the proof?

More subtle is the question of how much abstraction should be introduced into the specification. In the AAMP5 and AAMP-FV projects, we started with a register-transfer, or microarchitecture, abstraction of the processor. Even here, choices had to be made regarding how much detail should be included in the model. Including more detail makes it simpler to validate the model against the actual design but can have a profound impact on the complexity of the proofs. For example, in the AAMP5, abstract, property-oriented specifications of the bus interface unit (BIU) and look-ahead fetch unit (LFU) were created. While this saved considerable proof effort, the engineers were never as comfortable with these specifications as with the models that more closely followed the detailed design.

There are many choices to be made, and a complete set of proofs are not in themselves sufficient; one can have completely correct and consistent models that bear no relationship to reality. Ultimately, one has to honestly balance benefit against cost, organizing the effort so as to concentrate on the areas of highest risk. Even if every proof is not completed, it is usually possible to achieve a level of assurance in excess of what is accepted practice today.

On the AAMP-FV, we have chosen to create a model of the processor in PVS at the register-transfer level, to create a model of the processor at the instruction set level, and show that the register-transfer level model and the microcode correctly implement the instruction set specification. The most important proofs have been completed, although a few supporting lemmas have only been inspected and still need to be completed.

7.8 Support for Team Efforts

While the AAMP-FV team was small (three or four employees at Collins, one at SRI), it was large enough to encounter problems typically associated with team efforts. These included maintaining consistent configurations between all individuals, dividing work up among team members, and incorporating changes from one individual without adversely affecting other members of the team. All of these were compounded by having two development sites and by the fact that a PVS specification is a complex system consisting of a theories and proofs related through several dependencies.

The worst problems were simply those of maintaining a consistent configuration between team members. This was mitigated among the Collins team members by employing a commercial configuration management system to maintain independent views of the AAMP-FV specifications and proofs, integrate changes from one team member into the views of other team members, and save baselines. While the configuration management system was awkward to use and and seemed excessive for a project of moderate size, configuration control problems within the Collins site were minimal.

Unfortunately, this system was not available at SRI, and configuration control had to be performed manually. Normally, this consisted of sending individual theories and proof files between Collins and SRI. Inevitably, the Collins and SRI versions would drift apart as individuals forgot to send updated theories or proofs or neglected to install newly received changes to avoid conflicts with ongoing proofs. PVS provides a facility to generate a "dump" file that can be used to recreate

a PVS context, and such dumps were exchanged periodically. Unfortunately, no facilities exist for identifying the differences between a dump and an existing PVS context, or for extracting these differences and applying them to an existing context. As a result, team members were reluctant to switch to another's dump file for fear it would overlay their most recent changes.

Every few months, the failure to complete a complex proof would be traced, after many hours of work, to a discrepancy between the SRI and Collins configuration. Many more hours would then be devoted to "synchronizing" the specifications, which would immediately start to diverge again.

These problems can be addressed in a variety of ways, ranging from strict use of a manual protocol to implementation of a full, distributed, configuration management system. The most reasonable choice would be to use PVS with an existing configuration management system as was done on the AAMP-FV project. While this has worked well within Collins, a few minor problems have been observed. For example, when a file is checked out or checked in, PVS views it has being modified, and has to retypecheck all theories dependent on it. Care must be taken to check out and restore both the PVS theory (.pvs) and proof (.prf) files. Also, context (.bin) files pose a problem since they may be affected by a change to their context even though the associated theory or proof file has not been changed.

Distributed sites present a more complex problem. The ideal solution would be to use a configuration management system that supports distributed sites, but these are usually expensive. Since distributed team efforts using PVS are still rare, some simple facilities to check-in and check-out theories from a central site and to identify differences between two sites would probably be sufficient if combined with a manual protocol.

7.9 Use of Human Resources

It became clear on the AAMP-FV project that large formal verification efforts should be staffed with several levels of expertise. Some activities must be assigned to the most experienced individual, while others can be safely delegated to less experienced employees. Besides making good economic sense, this ensures that all team members remain challenged and helps ensure an increasing pool of skills.

Ideally, the most experienced individuals would be given the task of creating the PVS specifications, since there are numerous decisions that will affect the overall success of the project. Choices must be made as to what is to be modeled (i.e., what is the ultimate goal of the project), where abstraction can be used as opposed to simply copying the system design, and which styles of specification can be validated through informal reviews and which will best support the proofs. Some tasks, such as fleshing out specification details and conducting reviews, can be delegated to less experienced individuals, but the overall structure of the formal models must be directed by individuals with a good understanding of the problem domain and experience with both formal specification and verification.

Its also important to have experienced individuals involved with setting up the framework for the proofs and generating the first example proofs. While any completed proof is in some sense adequate, care needs to be taken to ensure that proofs are robust since changes to the specifications will occur even on a stable project. Also, an experienced individual can often come up with a far shorter and more elegant proof than a novice, and proofs should be efficient to minimize the cost of running them and their derivatives.

Once this framework is in place, actually completing the proofs can be delegated to the most inexperienced individuals on the team. One of the advantages of formal verification is that each proof provides a simple thumbs up or thumbs down, regardless of the quality of the proof. Even so, it would seem prudent to provide some sort of oversight to ensure that the proofs aren't drifting too far from the original robust and efficient examples devised by the more experienced team members.

Our experiences also suggest that at least one individual should be charged with responsibility for ensuring that different versions of the specifications remain synchronized and overseeing the integration of each team member's work into the overall specification and proofs. As with any project, it is also recommended that there be a project manager, project plan, and project schedule and that the project be tracked and managed against that plan.

Chapter 8

Conclusions and Future Directions

The central result of this project was to demonstrate that formal verification of microcode can be performed at reasonable cost for most of the AAMP-FV instruction set. We have formally specified in PVS the entire AAMP-FV microarchitecture, 54 of the AAMP-FV's 80 instructions, translated into PVS the microcode for these instructions. The microcode in these 54 instructions was proven correct except for the proofs of some supporting lemmas.[1] No errors were found in the microcode verified, although some mistakes were discovered in our specifications. The cost to verify these instructions was about 38 hours per instruction, almost an order of magnitude reduction over the AAMP5 costs.

Moreover, we are confident that the cost can be reduced further. Even on the AAMP-FV project, substantial effort went into refining and generalizing proof strategies. Much of this expertise can be reused on future efforts. Our current belief is that we could cut the cost per instruction to about one half our current costs, i.e., approximately 20 hours per instruction.

However, more work needs to be done on the verification of complex instructions such as multiplication, division, and procedure call and return. While techniques for verifying these instructions were developed on the AAMP-FV project, they have not been applied to all the remaining instructions. Verification of these instructions should eventually become routine, but they will probably always be more costly to prove correct than the simpler instructions. Unfortunately, we still do not have a good estimate of what the cost of verifying these instructions could be reduced to. More work needs to be done in exploring the variety of instructions to be verified and refining the strategies developed on the AAMP-FV project.

In retrospect, there has been a steady advancement of technology and techniques for formal verification and its transfer to Collins on both the AAMP5 and AAMP-FV projects. The AAMP5 project demonstrated how formal specifications could be applied to a complex, pipelined microprocessor and laid the groundwork for formal verification of microcode. On the AAMP-FV project, formal specification and verification of simple instructions were well understood and applied routinely. Techniques for the more complex instructions were developed and are now being transferred to Collins.

[1] In a later phase of the project, SRI completed the proofs of these supporting lemmas, as well as the proofs of some of the more complex instructions such as CALL and IMPY. These proofs were run top-to-bottom to ensure no lemma was left unproved in the proof chain. The axioms in the new specification have not been validated by Collins, but the proofs have been installed and executed by them. SRI also explored in this later phase ways to automate the proofs and make them more efficient. This work is documented in [33].

There are several other improvements that could be made. The AAMP-FV project has convinced us of the importance of sheer computing power when attempting formal verification on an industrial scale. The extensive rewriting capabilities and decision procedures of PVS played an essential role in completing the AAMP-FV proofs—without them the project would not have been feasible. Even so, considerable time is spent waiting on the prover, and still more speed would be helpful. On the AAMP-FV project, an enhancement to PVS that improved its performance by a factor of four made possible approaches that were previously infeasible simply because they took too long. Future improvements in workstation speed and enhancements to PVS will drop the man-hours per instruction further yet.

Another important area for improvement is in bit vector libraries. In hardware verification, the manipulation of bit vectors accounts for most of the verification effort. The bit vector libraries developed for the AAMP5 are extensive, but there are still questions of whether they are the best form needed to support both specification and proofs. The rewrite lemmas developed for the AAMP-FV project have moved us away from dependence on a specific bit vector representation, although proving their correctness using at least one model serves to demonstrate their consistency. Ideally, decision procedures for the bit vectors will be added directly to PVS to supplement those currently available for arithmetic equality and BDD-based Boolean simplification. Even so, rewrite rules for the bit vectors will still be needed to supplement the core functionality provided by the decision procedure.

The AAMP-FV project also made it clear that it is not sufficient simply to complete a proof once. Even though the design of the AAMP-FV was not changed during the project, the PVS specifications were changed frequently to correct errors and to simplify later proofs. All too often, these changes "broke" proofs that had already been "completed", that then had to be corrected at considerable expense. Techniques for making proofs more robust would have reduced both the cost and annoyance of making such changes. On most industrial projects, the requirements and design will be far less stable than for the AAMP-FV and the need for robust proofs considerably greater. This can be achieved both by engineering proofs to be more robust and by increasing the capabilities of PVS. One possibility is the development of more powerful prover commands to reduce the need for user involvement at low levels of detail. Improved user interfaces would also make it easier to see patterns and to alter proofs to make them more robust.

Another important way to reduce the cost of formal verification is to reuse specifications and proofs. Examples need to be done to explore how families of products can be specified. How can common features be specified and verified, and how can these be reused and extended to specify a single instance of the family?

A related question involves the reuse of proof strategies. Many of the micro correctness and macro lift proofs associated with the AAMP-FV are all quite similar. This is a common situation in industrial projects, where products are often built from variations on a few standard design patterns. Our reuse strategy was a simple one; we copied the proof of a similar instruction, edited it, ran it, then "tweaked" the proof until it completed. More sophisticated approaches would allow reuse of one copy of the core strategy, with the "tweaks" added as specializations of this strategy.

As formal verification evolves from small efforts staffed almost entirely by formal methods experts to industrial efforts staffed by domain experts, thought will need to be given to how the tools and management of formal verification will have to change. Configuration management of a complete PVS verification, including specifications, proofs, proof strategies, and context, can become complex even for a project the size of the AAMP5 and AAMP-FV. As teams grow, more

attention will have to be paid to how work is divided among individuals with different levels of expertise. What facilities will be needed to lower the entry level for novices? If environments for group verification are to be developed, what features should these environments have?

Formal verification is moving towards realistic use in industrial settings. There has been steady advancement and transfer of this technology to Collins. In fact, sufficient expertise has been developed that it is being used not only in the ways anticipated, but in unforeseen ways by real engineers on real projects. Further inroads will occur as tools, performance, and the pool of examples improve.

Acknowledgements—The authors thank Rick Butler and Paul Miner of NASA Langley for their support of the AAMP-FV and AAMP5 projects, Sam Owre and Natarjan Shankar of SRI International for the development and maintenance of PVS, and Al Mass of Rockwell International for translation of the microcode to PVS and consultation. We also thank John Rushby of SRI and John Gee, Mark Kovalan, Charlie Kress, Steve Maher, Mike Masters, Nick Mykris, Jeff Russell, and Roger Shultz of Rockwell for their support and assistance.

Bibliography

[1] Geoff Barrett. Formal methods applied to a floating-point number system. *IEEE Transactions on Software Engineering*, 15(5):611–621, May 1989.

[2] Derek L. Beatty and Randal E. Bryant. Formally verifying a microprocessor using a simulation methodology. In *Proceedings of the 31st Design Automation Conference*, pages 596–602. Association for Computing Machinery, June 1994.

[3] David W. Best, Charles E. Kress, Nick M. Mykris, Jeffrey D. Russell, and William J. Smith. An advanced-architecture CMOS/SOS microprocessor. *IEEE Micro*, 2(4):11–26, August 1982.

[4] William R. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1368–81, November 1989. Also published as CLI Technical Report 28.

[5] R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, New York, NY, 1979.

[6] Robert S. Boyer and Yuan Yu. Automated correctness proofs of machine code programs for a commercial microprocessor. In D. Kapur, editor, *Automated Deduction – CADE-11*, number 607 in Lecture Notes in Computer Science, pages 416–430. Springer-Verlag, 1992.

[7] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In David Dill, editor, *Computer-Aided Verification, CAV '94*, volume 818 of *Lecture Notes in Computer Science*, pages 68–80, Stanford, CA, June 1994. Springer-Verlag.

[8] R. Butler and G. Finelli. The infeasibility of experimental quantification of life-critical software reliability. *IEEE Transactions on Software Engineering*, 16(5):66–76, January 1993.

[9] Ricky W. Butler. NASA Langley's research program in formal methods. In *COMPASS '91 (Proceedings of the Sixth Annual Conference on Computer Assurance)*, pages 157–162, Gaithersburg, MD, June 1991. IEEE Washington Section.

[10] Ricky W. Butler, Paul S. Miner, Mandayam K. Srivas, Dave A. Greve, and Steven P. Miller. A bitvectors library for PVS. Technical Memorandum 110274, NASA, Langley Research Center, Hampton, VA, August 1996.

[11] W. C. Carter, W. H. Joyner, Jr., and D. Brand. Microprogram verification considered necessary. In *National Computer Conference*, volume 48, pages 657–664. AFIPS Conference Proceedings, 1978.

[12] J. V. Cook. Final report for the C/30 microcode verification project. Technical Report ATR-86(6771)-3, Computer Science Laboratory, The Aerospace Corporation, El Segundo, CA, September 1986.

[13] D. Cyrluk, S. Rajan, N. Shankar, and M. K. Srivas. Effective theorem proving for hardware verification. In Kumar and Kropf [24], pages 287–305.

[14] Ben L. Di Vito, Ricky W. Butler, and James L. Caldwell. Formal design and verification of a reliable computing platform for real-time control. NASA Technical Memorandum 102716, NASA Langley Research Center, Hampton, VA, October 1990.

[15] David W. Embley and Scott N. Woodfiled. Assessing the quality of abstract data types written in Ada. In *Tenth International Conference on Software Engineering*, pages 144–153. IEEE Computer Society Press, April 1988.

[16] Colin Fidge, Peter Kearney, and Mark Utting. Formal specification and interactive proof of a simple real-time scheduler. Technical Report 94-11, Software Verification Research Centre, The University of Queensland, April 1994.

[17] S. Gerhart, M. Bouler, K. Greene, D. Jamsek, T. Ralston, and D. Russinoff. Formal methods transition study final report. Technical Report STP-FT-322-91, Microelectronics and Computer Technology Corporation, Austin, Texas, August 1991.

[18] M. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. Technical Report 77, University of Cambridge Computer Laboratory, September 1985.

[19] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic.* Cambridge University Press, Cambridge, UK, 1993.

[20] Warren A. Hunt, Jr. *FM8501: A Verified Microprocessor*, volume 795 of *Lecture Notes in Artificial Intelligence.* Springer-Verlag, Berlin, 1994.

[21] Warren A. Hunt, Jr. and Bishop C. Brock. A formal HDL and its use in the FM9001 verification. In C. A. R. Hoare and M. J. C. Gordon, editors, *Mechanized Reasoning and Hardware Design*, pages 35–47, Hemel Hempstead, UK, 1992. Prentice Hall International Series in Computer Science.

[22] Jeffrey Joyce. Verification and implementation of a microprocessor. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis.* Kluwer Academic Publishers, Boston, MA, 1988.

[23] Peter Kearney and Mark Utting. A layered real-time specification of a RISC processor. In Costas Courcoubetis, editor, *Computer-aided Verification – CAV '93*, volume 697 of *Lecture Notes in Computer Science.* Springer-Verlag, 1993.

[24] Ramayya Kumar and Thomas Kropf, editors. *Preliminary Proceedings of the Second Conference on Theorem Provers in Circuit Design*, Bad Herrenalb (Blackforest), Germany, September 1994. Forschungszentrum Informatik an der Universität Karlsruhe, FZI Publication 4/94.

[25] George B. Leeman, William C. Carter, and Alexander Birman. Some techniques for microprogram validation. In *Information Processing 74 (Proc. IFIP Congress 1974)*, pages 76–80. North-Holland Publishing Co, 1974.

[26] B. Littlewood and L. Strigini. Validation of ultra-high dependability of software-based systems. *CACM*, November 1993.

[27] Steven P. Miller and Mandayam Srivas. Formal verification of the AAMP5 microprocessor: A case study in the industrial use of formal methods. In *WIFT'95: Workshop on Industrial-Strength Formal specification Techniques*, Boca Raton, FL, 1995. IEEE Computer Society.

[28] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[29] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[30] James B. Saxe, Stephen J. Garland, John V. Guttag, and James J. Horning. Using transformations and verification in circuit design. *Formal Methods in System Design*, 4(1):181–210, 1994.

[31] N. Shankar, S. Owre, and J. M. Rushby. *The PVS Proof Checker: A Reference Manual (Beta Release).* Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.

[32] Mary Shaw. Prospects for an engineering discipline of software. *IEEE Software*, 7(11):15–24, November 1990.

[33] Mandayam Srivas. Automating microcode verification via symbolic simulation: The AAMP-FV experiment. Contractor report, NASA Langley Research Center, Hampton, VA. (Forthcoming).

[34] Mandayam Srivas and Mark Bickford. Formal verification of a pipelined microprocessor. *IEEE Software*, 7(5):52–64, September 1990.

[35] Mandayam Srivas and Steven P. Miller. Applying formal verification to a commercial microprocessor. In *IFIP Conference on Hardware Description Languages and Their Applications (CHDL'95)*, Makuhari, Chiba, Japan, August 1995.

[36] Mandayam Srivas and Steven P. Miller. Formal verification of an avionics microprocessor. Technical Report NASA Contractor Report 4682, NASA Langley Research Center, Hampton, Virginia, July 1995.

[37] Mandayam Srivas and Steven P. Miller. Formal verification of the AAMP5 microprocessor. In Jonathan P. Bowen and Michael G. Hinchey, editors, *Applications of Formal Methods.* Prentice-Hall International Ltd., Hemel Hempstead, UK, 1995.

[38] Mandayam K. Srivas and Steven P. Miller. Applying formal verification to the AAMP5 microprocessor: A case study in the industrial use of formal methods. *Formal Methods in Systems Design*, 8(2):153–188, March 1996.

[39] Matthew Wilding. A mechanically verified application for a mechanically verified environment. In Costas Courcoubetis, editor, *Computer-aided Verification – CAV '93*, volume 697 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[40] Matthew Wilding. *Machine-Checked Real-Time System Verification.* PhD thesis, University of Texas at Austin, 1996.

[41] Phillip J. Windley. *The Formal Verification of Generic Interpreters.* PhD thesis, University of California, Davis, CA, June 1990.

[42] Phillip J. Windley and Michael L. Coe. A correctness model for pipelined microprocessors. In Kumar and Kropf [24], pages 35–54.

REPORT DOCUMENTATION

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	February 1999	Contractor Report

4. TITLE AND SUBTITLE

Formal Verification of the AAMP-FV Microcode

5. FUNDING NUMBERS

C NAS1-19704, NAS1-20334

WU 519-30-31-01

6. AUTHOR(S)

Steven P. Miller, David A. Greve, Matthew M. Wilding,
Mandayam Srivas

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Rockwell Collins, Inc.; Cedar Rapids, IA

SRI International; Menlo Park, CA

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23681-2199

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

NASA/CR-1999-208992

11. SUPPLEMENTARY NOTES

Miller, Greve, Wilding: Rockwell Collins, Cedar Rapids, IA; Srivas: SRI International, Menlo Park, CA.
Langley Technical Monitor: Paul S. Miner Final Report

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Unclassified-Unlimited
Subject Category 61 Distribution: Standard
Availability: NASA CASI, (301) 621-0390

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This report describes the experiences of Collins Avionics & Communications and SRI International in formally specifying and verifying the microcode in a Rockwell proprietary microprocessor, the AAMP-FV, using the PVS verification system. This project built extensively on earlier experiences using PVS to verify the microcode in the AAMP5, a complex, pipelined microprocessor designed for use in avionics displays and global positioning systems. While the AAMP5 experiment demonstrated the technical feasibility of formal verification of microcode, the steep learning curve encountered left unanswered the question of whether it could be performed at reasonable cost. The AAMP-FV project was conducted to determine whether the experience gained on the AAMP5 project could be used to make formal verification of microcode cost effective for safety-critical and high volume devices.

14. SUBJECT TERMS

Formal Methods, Microcode Verification, Theorem Proving

15. NUMBER OF PAGES

72

16. PRICE CODE

A04

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	